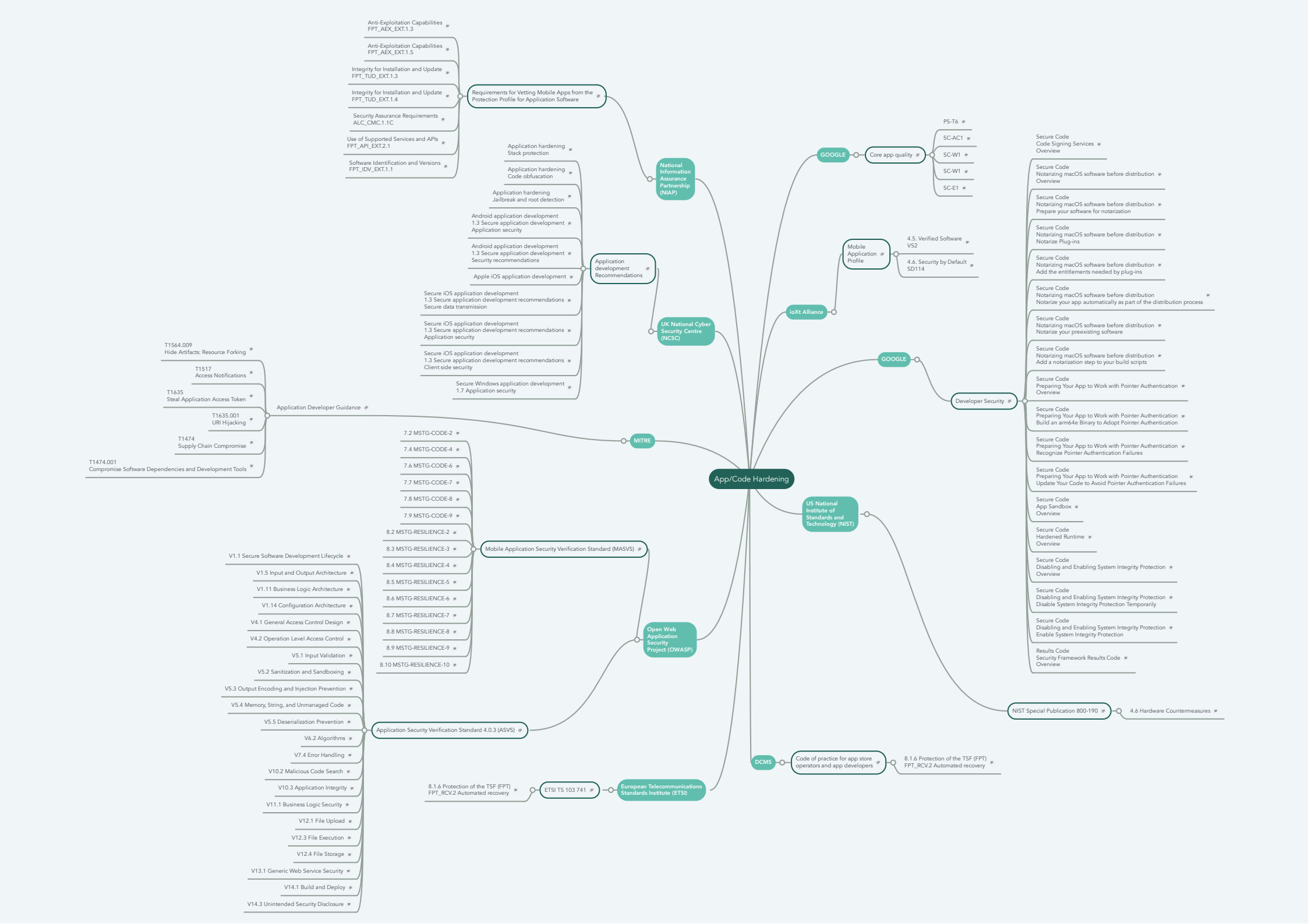
App/Code Hardening
National Information Assurance Partnership (NIAP)
Requirements for Vetting Mobile Apps from the Protection Profile for Application Software
Anti-Exploitation Capabilities FPT_AEX_EXT.1.3
Anti-Exploitation Capabilities FPT_AEX_EXT.1.5
Integrity for Installation and Update FPT_TUD_EXT.1.3
Integrity for Installation and Update FPT_TUD_EXT.1.4
Security Assurance Requirements ALC_CMC.1.1C
Use of Supported Services and APIs FPT_API_EXT.2.1
Software Identification and Versions FPT_IDV_EXT.1.1
UK National Cyber Security Centre (NCSC)
Application development Recommendations
Application hardening Stack protection
Application hardening Code obfuscation
Application hardening Jailbreak and root detection
Android application development 1.3 Secure application development Application security
Android application development 1.3 Secure application development Security recommendations
Apple iOS application development
Secure iOS application development 1.3 Secure application development recommendations Secure data transmission
Secure iOS application development 1.3 Secure application development recommendations Application security
Secure iOS application development 1.3 Secure application development recommendations Client side security
Secure Windows application development 1.7 Application security
MITRE
Application Developer Guidance
T1564.009 Hide Artifacts: Resource Forking
T1517 Access Notifications
T1635 Steal Application Access Token
T1635.001 URI Hijacking
T1474 Supply Chain Compromise
T1474.001 Compromise Software Dependencies and Development Tools
Open Web Application Security Project (OWASP)
Mobile Application Security Verification Standard (MASVS)
7.2 MSTG-CODE-2
7.4 MSTG-CODE-4
7.6 MSTG-CODE-6
7.7 MSTG-CODE-7
7.8 MSTG-CODE-8
7.9 MSTG-CODE-9
8.2 MSTG-RESILIENCE-2
8.3 MSTG-RESILIENCE-3
8.4 MSTG-RESILIENCE-4
8.5 MSTG-RESILIENCE-5
8.6 MSTG-RESILIENCE-6
8.7 MSTG-RESILIENCE-7
8.8 MSTG-RESILIENCE-8
8.9 MSTG-RESILIENCE-9
8.10 MSTG-RESILIENCE-10
Application Security Verification Standard 4.0.3 (ASVS)
V1.1 Secure Software Development Lifecycle
V1.5 Input and Output Architecture
V1.11 Business Logic Architecture
V1.14 Configuration Architecture
V4.1 General Access Control Design
V4.2 Operation Level Access Control
V5.1 Input Validation
V5.2 Sanitization and Sandboxing
V5.3 Output Encoding and Injection Prevention
V5.4 Memory, String, and Unmanaged Code
V5.5 Deserialization Prevention
V6.2 Algorithms
V7.4 Error Handling
V10.2 Malicious Code Search
V10.3 Application Integrity
V11.1 Business Logic Security
V12.1 File Upload
V12.3 File Execution
V12.4 File Storage
V13.1 Generic Web Service Security
V14.1 Build and Deploy
V14.3 Unintended Security Disclosure
European Telecommunications Standards Institute (ETSI)
ETSI TS 103 741
8.1.6 Protection of the TSF (FPT) FPT_RCV.2 Automated recovery
DCMS
Code of practice for app store operators and app developers
8.1.6 Protection of the TSF (FPT) FPT_RCV.2 Automated recovery
US National Institute of Standards and Technology (NIST)
NIST Special Publication 800-190
4.6 Hardware Countermeasures
GOOGLE
Developer Security
Secure Code Code Signing Services Overview
Secure Code Notarizing macOS software before distribution Overview
Secure Code Notarizing macOS software before distribution Prepare your software for notarization
Secure Code Notarizing macOS software before distribution Notarize Plug-ins
Secure Code Notarizing macOS software before distribution Add the entitlements needed by plug-ins
Secure Code Notarizing macOS software before distribution Notarize your app automatically as part of the distribution process
Secure Code Notarizing macOS software before distribution Notarize your preexisting software
Secure Code Notarizing macOS software before distribution Add a notarization step to your build scripts
Secure Code Preparing Your App to Work with Pointer Authentication Overview
Secure Code Preparing Your App to Work with Pointer Authentication Build an arm64e Binary to Adopt Pointer Authentication
Secure Code Preparing Your App to Work with Pointer Authentication Recognize Pointer Authentication Failures
Secure Code Preparing Your App to Work with Pointer Authentication Update Your Code to Avoid Pointer Authentication Failures
Secure Code App Sandbox Overview
Secure Code Hardened Runtime Overview
Secure Code Disabling and Enabling System Integrity Protection Overview
Secure Code Disabling and Enabling System Integrity Protection Disable System Integrity Protection Temporarily
Secure Code Disabling and Enabling System Integrity Protection Enable System Integrity Protection
Results Code Security Framework Results Code Overview
GOOGLE
Core app quality
PS-T6
SC-AC1
SC-W1
SC-W1
SC-E1
ioXt Alliance
Mobile Application Profile
4.5. Verified Software VS2
4.6. Security by Default SD114

App/Code Hardening

1. GOOGLE

1.1. Core app quality

Link: https://developer.android.com/docs/quality-guidelines/core-app-quality

1.1.1. PS-T6

No debug libraries are included in the production app. This can cause performance as well as security issues.

1.1.2. SC-AC1

The app sets the android:exported attribute explicitly for all activities, services, broadcast receivers, and especially content providers.

Only application components that share data with other apps, or components that should be invoked by other apps, are exported.

1.1.3. SC-W1

Do not use setAllowUniversalAccessFromFileURLs() for accessing local content. Instead, use WebViewAssetLoader.

1.1.4. SC-W1

WebViews should not use addJavaScriptInterface() with untrusted content.
On Android 6.0 and above, use HTML message channels instead.

1.1.5. SC-E1

The app does not dynamically load code from outside the app's APK. Developers should use Android App Bundles, which includes Play Feature Delivery and Play Asset Delivery.
Starting August 2021, the use of Android App Bundles will become mandatory for all new apps in the Google Play store.

2. ioXt Alliance

2.1. Mobile Application Profile

Link: https://static1.squarespace.com/static/5c6dbac1f8135a29c7fbb621/t/604aa3fa668a8e3b50630433/1615504379349/Mobile_Application_Profile.pdf

2.1.1. 4.5. Verified Software VS2

> Software images including plug-ins and apps are signed and verified.

2.1.2. 4.6. Security by Default SD114

> Use default compiler security flags.

3. GOOGLE

3.1. Developer Security

Link: https://developer.apple.com/documentation/security

3.1.1. Secure Code Code Signing Services Overview

> Code signing is a macOS security technology that you use to certify that an app was created by you. Once an app is signed, the system can detect any change to the app—whether the change is introduced accidentally or by malicious code. You can control how your signed code loads signed plug-ins and other signed code without invalidating the signatures of the host code or of the guest (dynamically loaded) code.
>
> You work with code objects that represent uniquely identified elements of running code in the system. In addition to UNIX processes, these elements can include scripts, applets, widgets, and so forth. You also work with static code objects that represent code in the file system. Static code includes applications, tools, frameworks, plug-ins, scripts, and so on. Generally, a code object has a specific static code object from which it originates and that holds its static signing data. The reverse, however, is not true—given a static code object, it is not possible to find, enumerate, or control any code object that originated from it.

3.1.2. Secure Code Notarizing macOS software before distribution Overview

Notarization gives users more confidence that the Developer ID-signed software you distribute has been checked by Apple for malicious components. Notarization is not App Review. The Apple notary service is an automated system that scans your software for malicious content, checks for code-signing issues, and returns the results to you quickly. If there are no issues, the notary service generates a ticket for you to staple to your software; the notary service also publishes that ticket online where Gatekeeper can find it.

When the user first installs or runs your software, the presence of a ticket (either online or attached to the executable) tells Gatekeeper that Apple notarized the software. Gatekeeper then places descriptive information in the initial launch dialog to help the user make an informed choice about whether to launch the app.

Screenshot of the dialog that Gatekeeper presents to let the user know that Apple notarized the app being launched.

You can notarize several different types of software deliverables, including:

macOS apps

Non-app bundles, such as kernel extensions

Disk images (UDIF format)

Flat installer packages

Notarization also protects your users if your Developer ID signing key is exposed. The notary service maintains an audit trail of the software distributed using your signing key. If you discover unauthorized versions of your software, you can work with Apple to revoke the tickets associated with those versions.

3.1.3. Secure Code Notarizing macOS software before distribution Prepare your software for notarization

Notarization requires Xcode 10 or later. Building a new app for notarization requires macOS 10.13.6 or later. Stapling an app requires macOS 10.12 or later.

Apple's notary service requires you to adopt the following protections:

Enable code-signing for all of the executables you distribute, and ensure that executables have valid code signatures, as described in Ensure a valid code signature.

Use a “Developer ID” application, kernel extension, system extension, or installer certificate for your code-signing signature. (Don't use a Mac Distribution, ad hoc, Apple Developer, or local development certificate.) Verify the certificate type before submitting, as described in Use a valid Developer ID certificate. For more information, see Create, export, and delete signing certificates.

Enable the Hardened Runtime capability for your app and command line targets, as described in Enable hardened runtime.

Include a secure timestamp with your code-signing signature. (The Xcode distribution workflow includes a secure timestamp by default. For custom workflows, see Include a secure timestamp.)

Don’t include the com.apple.security.get-task-allow entitlement with the value set to any variation of true. If your software hosts third-party plug-ins and needs this entitlement to debug the plug-in in the context of a host executable, see Avoid the get-task-allow entitlement.

Link against the macOS 10.9 or later SDK, as described in Use the macOS 10.9 SDK or later.

Ensure your processes have properly-formatted XML, ASCII-encoded entitlements as described in Ensure properly formatted entitlements.

Apple recommends that you notarize all of the software that you’ve distributed, including older releases, and even software that doesn’t meet all of these requirements or that is unsigned. Apple’s notary service uses a variety of methods, including telemetry, to determine which of the above rules to relax for preexisting software. For more information, see Notarize your preexisting software.

3.1.4. Secure Code Notarizing macOS software before distribution Notarize Plug-ins

In macOS 10.15 and later, apps can load quarantined plug-ins — those downloaded from the internet or transferred with AirDrop — only if the plug-in is notarized. For plug-ins that aren’t notarized, the user must explicitly approve the plug-in by opening the System Preferences app and navigating to the General tab in Security & Privacy.

3.1.5. Secure Code Notarizing macOS software before distribution Add the entitlements needed by plug-ins

When you enable the extra security enforced by the hardened runtime, as notarization requires, this impacts both your app and any plug-ins that your app hosts. Plug-ins don’t declare their own entitlements. Instead, they inherit the entitlements of the host process. Therefore, a host app must include all the entitlements that prospective plug-ins require, even when the plug-ins are notarized separately.

For example, if a plug-in employs deep integration with the host executable via C function pointer overrides, or uses a JavaScript engine for custom workflows, the host executable must declare the Allow Unsigned Executable Memory Entitlement or Allow Execution of JIT-compiled Code Entitlement, respectively. In some cases, a plug-in fails to even load if the host executable lacks the proper entitlement.

Be aware that even if your app doesn’t provide a dedicated plug-in architecture, it might still load plug-ins, like drivers for professional mirrorless cameras and legacy DSLR cameras that don’t conform to the driverless USB video device class (UVC) standard. If your app works with this kind of hardware, be sure to declare the Disable Library Validation Entitlement to load the corresponding plug-ins.

Also include resource access entitlements, like the Address Book or Location access entitlements, and the related purpose strings, that support your app’s plug-ins. For example, if a Print Dialog Extension (PDE) that provides fax services wants to access a user’s contact list, the host executable must declare the Address Book Entitlement and include the NSContactsUsageDescription purpose string in its Information Property List for the plug-in to operate.

For a complete list of hardened runtime entitlements, see Hardened Runtime. For information about usage strings, see Requesting Access to Protected Resources.

3.1.6. Secure Code Notarizing macOS software before distribution Notarize your app automatically as part of the distribution process

Before distributing your app directly to customers, your Account Holder must sign the app with your Developer ID. Xcode’s Organizer window includes a workflow for generating a distributable version of your app. This workflow includes an option to notarize your macOS app automatically. To notarize your app using this workflow, do the following:

Open your Xcode project.

Create an archive of your app.

Open Xcode's Organizer window.

In the Archives tab, select the archive you created.

Click Distribute App to view the distribution options.

Choose Developer ID for your method of distribution.

Click Next.

Choose Upload to send your archive to the Apple notary service.

Click Next.

Screenshot of Xcode's Organizer window showing the Upload and Export destination options for a given archive.

When you click Next, Xcode uploads your archive to the notary service. When the upload is complete, the notary service begins the scanning process, which usually takes less than an hour. While the notary service scans your software, you can continue to prepare your archive for distribution. For example, you can export the archive and perform any final testing that you require prior to making your software available to customers.

When the notarization process finishes, Xcode downloads the ticket and staples it to your archive. At that point, export your archive again to receive a distributable version of your software that includes the notary ticket. For more information about how to use the Xcode UI to upload your software, see Upload a macOS app to be notarized.

If Xcode doesn’t let you upload for notarization, be sure that you are building a macOS archive, as described in Notarize macOS apps with external dependencies. For targets other than macOS apps, use the notarytool command line utility to notarize, as described in Customizing the notarization workflow.

3.1.7. Secure Code Notarizing macOS software before distribution Notarize your preexisting software

Notarizing your preexisting software lets Gatekeeper warn users when they try to run it. It also helps the notary service distinguish your legitimate software from variants that have been tampered with. You can notarize an existing disk image, installer package, or ZIP archive containing your app.

To notarize your preexisting software, do the following:

Make your active Xcode installation one that supports notarization by using the xcode-select command-line tool. For information about how to use this tool, see its man page, as described in Reading UNIX Manual Pages.

Upload your software to the Apple notary service, as described in Upload your app to the notarization service.

Staple the returned ticket to your existing software, as described in Staple the ticket to your distribution.

3.1.8. Secure Code Notarizing macOS software before distribution Add a notarization step to your build scripts

If you use an automated build system, you can integrate the notarization process into your existing build scripts. The notarytool and stapler command-line tools (included with Xcode) allow you to upload your software to the Apple notary service, and to staple the resulting ticket to your executable. Alternatively, you can interact directly with the notary service using the Notary API.

For information about how to incorporate notarization into your custom build scripts, see Customizing the notarization workflow.

3.1.9. Secure Code Preparing Your App to Work with Pointer Authentication Overview

The arm64e architecture introduces pointer authentication codes (PACs) to detect and guard against unexpected changes to pointers in memory. The addition of pointer authentication is transparent to most apps because the compiler manages the process. In rare cases — for example, if your app manipulates the stack directly, or if you pass pointers between C++ and Objective-C++ — you might have to adjust your code to work with PACs.

Pointer authentication works by offering a special CPU instruction to add a cryptographic signature — or PAC — to unused high-order bits of a pointer before storing the pointer. Another instruction removes and authenticates the signature after reading the pointer back from memory. Any change to the stored value between the write and the read invalidates the signature. The CPU interprets authentication failure as memory corruption and sets a high-order bit in the pointer, making the pointer invalid and causing the app to crash

3.1.10. Secure Code Preparing Your App to Work with Pointer Authentication Build an arm64e Binary to Adopt Pointer Authentication

You automatically adopt pointer authentication in your app when you build and deploy a binary that targets the arm64e architecture. You can do this starting in Xcode 10.1. To build an arm64e slice, go to your iOS target's build settings in Xcode and find the Architectures item. Click the current setting and choose Other. In the box that appears, add arm64e.

Screenshot of Xcode showing the addition of arm64e to the Architectures item in the Build Settings pane for the iOS app target.

Devices using the Apple A12 or later A-series processor — like the iPhone XS, iPhone XS Max, and iPhone XR — support the arm64e architecture. To test your adoption, you have to run your app on one of these devices. You can't test using the Simulator.

3.1.11. Secure Code Preparing Your App to Work with Pointer Authentication Recognize Pointer Authentication Failures

When pointer authentication fails, the system invalidates the failing pointer by setting a high-order bit. Subsequent use of the pointer results in a segmentation fault. The crash report contains a message that includes the value of the pointer both after and before invalidation:

Exception Subtype: KERN_INVALID_ADDRESS at 0x0040000105394398 -> 0x0000000105394398 (possible pointer authentication failure)
Typically, the compiler adds the CPU instructions for both creating and authenticating the PAC. In the unusual case that you manage these steps yourself — for example, if you're authoring your own compiler — and if you try to use a signed pointer without first applying the authentication instruction to remove the signature, that also triggers a segmentation fault. In this case, the presence of the signature in the high-order bits invalidates the pointer:

Exception Subtype: KERN_INVALID_ADDRESS at 0x217c000105394398 -> 0x0000000105394398 (possible pointer authentication failure)
Be aware that other invalid memory accesses, where high-order bits are erroneously set, can also look like pointer authentication failures.

3.1.12. Secure Code Preparing Your App to Work with Pointer Authentication Update Your Code to Avoid Pointer Authentication Failures

Most code doesn’t require modification to run with pointer authentication, with the possible exception of some low-level code that relies on arm64-specific behavior. For example, a crash reporting library that examines the stack contents needs to strip the PAC out of return addresses. The Apple Clang compiler provides utilities in the ptrauth.h header file — like the ptrauth_strip macro — to help with these kinds of tasks.

Pointer authentication can also expose latent bugs in existing code. In C++, it’s incorrect to call a virtual method using a declaration that differs from its definition. In practice, such calls typically succeed in arm64, but trigger a pointer authentication failure in arm64e. You might encounter this bug when using OS_OBJECT types like dispatch_queue_t and xpc_connection_t. You can’t pass instances of these types from C++ code to an Objective-C++ function (or vice versa) because they’re defined differently in Objective-C++ to support automatic reference counting (ARC).

More generally, the PAC calculation takes into account the pointer value, one of several keys loaded into the CPU, and an optional salt value. To prevent reuse of pointers in different contexts, the PAC calculation depends on the pointer type. Keep these rules in mind when looking for possible issues in your code:

Return addresses are signed with a key that’s unique per process, using a salt derived from the stack pointer.

Function pointers are signed with a key that’s fixed across all processes, allowing sharing of library code between processes.

Virtual method table entries are signed with a key that’s shared across all apps, using a salt derived from the method signature.

3.1.13. Secure Code App Sandbox Overview

App Sandbox provides protection to system resources and user data by limiting your app’s access to resources requested through entitlements.

3.1.14. Secure Code Hardened Runtime Overview

The Hardened Runtime, along with System Integrity Protection (SIP), protects the runtime integrity of your software by preventing certain classes of exploits, like code injection, dynamically linked library (DLL) hijacking, and process memory space tampering. To enable the Hardened Runtime for your app, navigate in Xcode to your target's Signing & Capabilities information and click the + button. In the window that appears, choose Hardened Runtime.

Screenshot highlighting where to click to add a new capability in Xcode's Signing & Capabilities tab.

The Hardened Runtime doesn't affect the operation of most apps, but it does disallow certain less common capabilities, like just-in-time (JIT) compilation. If your app relies on a capability that the Hardened Runtime restricts, add an entitlement to disable an individual protection. You add an entitlement by enabling one of the runtime exceptions or access permissions listed in Xcode. Make sure to use only the entitlements that are absolutely necessary for your app's functionality.

Screenshot of Xcode showing some of the entitlements used for exceptions to the Hardened Runtime.

You add entitlements only to executables. Shared libraries, frameworks, and in-process plug-ins inherit the entitlements of their host executable.

The default value of these Boolean entitlements is false. When Xcode signs your code, it includes an entitlement only if the value is true. If you're manually signing code, follow this convention to ensure maximum compatibility. Don't include an entitlement if the value is false.

3.1.15. Secure Code Disabling and Enabling System Integrity Protection Overview

System Integrity Protection (SIP) in macOS protects the entire system by preventing the execution of unauthorized code. The system automatically authorizes apps that the user downloads from the App Store. The system also authorizes apps that a developer notarizes and distributes directly to users. The system prevents the launching of all other apps by default.

During development, it may be necessary for you to disable SIP temporarily to install and test your code. You don't need to disable SIP to run and debug apps from Xcode, but you might need to disable it to install system extensions, such as DriverKit drivers.

3.1.16. Secure Code Disabling and Enabling System Integrity Protection Disable System Integrity Protection Temporarily

To disable SIP, do the following:

Restart your computer in Recovery mode.

Launch Terminal from the Utilities menu.

Run the command csrutil disable.

Restart your computer.

3.1.17. Secure Code Disabling and Enabling System Integrity Protection Enable System Integrity Protection

To reenable SIP, do the following:

Restart your computer in Recovery mode.

Launch Terminal from the Utilities menu.

Run the command csrutil enable.

Restart your computer.

3.1.18. Results Code Security Framework Results Code Overview

Use the SecCopyErrorMessageString(_:_:) function to obtain a human readable string corresponding to these status codes.

In addition to the codes listed here, certain Security framework services provide additional status codes that are specific to that service. In particular, see Authorization Services Result Codes, Sessions API Result Codes, Secure Transport Result Codes, Secure Download Result Codes, and Code Signing Services Result Codes.

4. US National Institute of Standards and Technology (NIST)

4.1. NIST Special Publication 800-190

Link: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-190.pdf

4.1.1. 4.6 Hardware Countermeasures

Software-based security is regularly defeated, as acknowledged in NIST SP 800-164 [24]. NIST defines trusted computing requirements in NIST SPs 800-147 [25], 800-155 [26], and 800-164. To NIST, “trusted” means that the platform behaves as it is expected to: the software inventory is accurate, the configuration settings and security controls are in place and operating as they should, and so on. “Trusted” also means that it is known that no unauthorized person has tampered with the software or its configuration on the hosts. Hardware root of trust is not a concept unique to containers, but container management and security tools can leverage attestations for the rest of the container technology architecture to ensure containers are being run in secure environments.

The currently available way to provide trusted computing is to:
1. Measure firmware, software, and configuration data before it is executed using a Root of Trust for Measurement (RTM).
2. Store those measurements in a hardware root of trust, like a trusted platform module (TPM).
3. Validate that the current measurements match the expected measurements. If so, it can be attested that the platform can be trusted to behave as expected.

TPM-enabled devices can check the integrity of the machine during the boot process, enabling protection and detection mechanisms to function in hardware, at pre-boot, and in the secure boot process. This same trust and integrity assurance can be extended beyond the OS and the boot loader to the container runtimes and apps. Note that while standards are being developed to enable verification of hardware trust by users of cloud services, not all clouds expose this functionality to their customers. In cases where technical verification is not provided, organizations should address hardware trust requirements as part of their service agreements with cloud providers.

The increasing complexity of systems and the deeply embedded nature of today’s threats means that security should extend across all container technology components, starting with the hardware and firmware. This would form a distributed trusted computing model and provide the most trusted and secure way to build, run, orchestrate, and manage containers.

The trusted computing model should start with measured/secure boot, which provides a verified system platform, and build a chain of trust rooted in hardware and extended to the bootloaders, the OS kernel, and the OS components to enable cryptographic verification of boot mechanisms, system images, container runtimes, and container images. For container technologies, these techniques are currently applicable at the hardware, hypervisor, and host OS layers, with early work in progress to apply these to container-specific components. As of this writing, NIST is collaborating with industry partners to build reference architectures based on commercial off-the-shelf products that demonstrate the trusted computing model for container environments

5. DCMS

5.1. Code of practice for app store operators and app developers

Link: https://www.gov.uk/government/publications/code-of-practice-for-app-store-operators-and-app-developers/code-of-practice-for-app-store-operators-and-app-developers

5.1.1. 8.1.6 Protection of the TSF (FPT) FPT_RCV.2 Automated recovery

> 6.2 App Store Operators should publicise any upcoming changes to be introduced to their Developer guidelines / policies.

6. European Telecommunications Standards Institute (ETSI)

6.1. ETSI TS 103 741

Link: https://www.etsi.org/deliver/etsi_ts/103700_103799/103732/01.01.01_60/ts_103732v010101p.pdf

6.1.1. 8.1.6 Protection of the TSF (FPT) FPT_RCV.2 Automated recovery

> FPT_RCV.2.1 When automated recovery from detection of a malevolent persistent presence by FPT_TST.1 is not possible, the TSF shall enter a maintenance mode where the ability to return to a secure state is provided.
> FPT_RCV.2.2 For detection of a malevolent persistent presence by FPT_TST.1, the TSF shall ensure the return of the TOE to a secure state using automated procedures.
> NOTE 9: FPT_RCV.2.2 mandates that the TOE returns to a secure state using automated procedures. This state can be one where the malevolent persistent presence is completely removed, or it can be a state where the malevolent persistent presence is not loaded, or otherwise not activated. In the case where it is not possible to automatically remove the malevolent persistent presence, FPT_RCV.2.1 allows human user to remove it manually, for example by a factory reset or download of an update.

7. Open Web Application Security Project (OWASP)

7.1. Mobile Application Security Verification Standard (MASVS)

Link: https://github.com/OWASP/owasp-masvs/releases/tag/v1.4.2

7.1.1. 7.2 MSTG-CODE-2

> The app has been built in release mode, with settings appropriate for a release build (e.g. non-debuggable).

7.1.2. 7.4 MSTG-CODE-4

Debugging code and developer assistance code (e.g. test code, backdoors, hidden settings) have been removed. The app does not log verbose errors or debugging messages.

7.1.3. 7.6 MSTG-CODE-6

The app catches and handles possible exceptions.

7.1.4. 7.7 MSTG-CODE-7

Error handling logic in security controls denies access by default.

7.1.5. 7.8 MSTG-CODE-8

In unmanaged code, memory is allocated, freed and used securely.

7.1.6. 7.9 MSTG-CODE-9

Free security features offered by the toolchain, such as byte-code minification, stack protection, PIE support and automatic reference counting, are activated.

7.1.7. 8.2 MSTG-RESILIENCE-2

The app prevents debugging and/or detects, and responds to, a debugger being attached. All available debugging protocols must be covered.

7.1.8. 8.3 MSTG-RESILIENCE-3

The app detects, and responds to, tampering with executable files and critical data within its own sandbox.

7.1.9. 8.4 MSTG-RESILIENCE-4

The app detects, and responds to, the presence of widely used reverse engineering tools and frameworks on the device.

7.1.10. 8.5 MSTG-RESILIENCE-5

The app detects, and responds to, being run in an emulator.

7.1.11. 8.6 MSTG-RESILIENCE-6

The app detects, and responds to, tampering the code and data in its own memory space.

7.1.12. 8.7 MSTG-RESILIENCE-7

The app implements multiple mechanisms in each defense category (8.1 to 8.6). Note that resiliency scales with the amount, diversity of the originality of the mechanisms used.

7.1.13. 8.8 MSTG-RESILIENCE-8

The detection mechanisms trigger responses of different types, including delayed and stealthy responses.

7.1.14. 8.9 MSTG-RESILIENCE-9

Obfuscation is applied to programmatic defenses, which in turn impede de-obfuscation via dynamic analysis.

7.1.15. 8.10 MSTG-RESILIENCE-10

The app implements a ‘device binding’ functionality using a device fingerprint derived from multiple properties unique to the device.

7.2. Application Security Verification Standard 4.0.3 (ASVS)

Link: https://raw.githubusercontent.com/OWASP/ASVS/v4.0.3/4.0/OWASP%20Application%20Security%20Verification%20Standard%204.0.3-en.pdf

7.2.1. V1.1 Secure Software Development Lifecycle

1.1.1 Verify the use of a secure software development lifecycle that addresses security in all stages of development.

1.1.3 Verify that all user stories and features contain functional security constraints, such as "As a user, I should be able to view and edit my profile. I should not be able to view or edit anyone else's profile"

1.1.6 Verify implementation of centralized, simple (economy of design), vetted, secure, and reusable security controls to avoid duplicate, missing, ineffective, or insecure controls.

1.1.7 Verify availability of a secure coding checklist, security requirements, guideline, or policy to all developers and testers.

7.2.2. V1.5 Input and Output Architecture

1.5.3 Verify that input validation is enforced on a trusted service layer.

1.5.4 Verify that output encoding occurs close to or by the interpreter for which it is intended.

7.2.3. V1.11 Business Logic Architecture

1.11.2 Verify that all high-value business logic flows, including authentication, session management and access control, do not share unsynchronized state.

1.11.3 Verify that all high-value business logic flows, including authentication, session management and access control are thread safe and resistant to time-of-check and time-of-use race conditions.

7.2.4. V1.14 Configuration Architecture

1.14.4 Verify that the build pipeline contains a build step to automatically build and verify the secure deployment of the application, particularly if the application infrastructure is software defined, such as cloud environment build scripts.

1.14.5 Verify that application deployments adequately sandbox, containerize and/or isolate at the network level to delay and deter attackers from attacking other applications, especially when they are performing sensitive or dangerous actions such as deserialization.

7.2.5. V4.1 General Access Control Design

4.1.5 Verify that access controls fail securely including when an exception occurs.

7.2.6. V4.2 Operation Level Access Control

4.2.1 Verify that sensitive data and APIs are protected against Insecure Direct Object Reference (IDOR) attacks targeting creation, reading, updating and deletion of records, such as creating or updating someone else's record, viewing everyone's records, or deleting all records.

4.2.2 Verify that the application or framework enforces a strong anti-CSRF mechanism to protect authenticated functionality, and effective anti-automation or anti-CSRF protects unauthenticated functionality.

7.2.7. V5.1 Input Validation

5.1.1 Verify that the application has defenses against HTTP parameter pollution attacks, particularly if the application framework makes no distinction about the source of request parameters (GET, POST, cookies, headers, or environment variables).

5.1.2 Verify that frameworks protect against mass parameter assignment attacks, or that the application has countermeasures to protect against unsafe parameter assignment, such as marking fields private or similar.

5.1.3 Verify that all input (HTML form fields, REST requests, URL parameters, HTTP headers, cookies, batch files, RSS feeds, etc) is validated using positive validation (allow lists).

5.1.4 Verify that structured data is strongly typed and validated against a defined schema including allowed characters, length and pattern (e.g. credit card numbers, e-mail addresses, telephone numbers, or validating that two related fields are reasonable, such as checking that suburb and zip/postcode match).

5.1.5 Verify that URL redirects and forwards only allow destinations which appear on an allow list, or show a warning when redirecting to potentially untrusted content.

7.2.8. V5.2 Sanitization and Sandboxing

5.2.1 Verify that all untrusted HTML input from WYSIWYG editors or similar is properly sanitized with an HTML sanitizer library or framework feature.

5.2.2 Verify that unstructured data is sanitized to enforce safety measures such as allowed characters and length.

5.2.3 Verify that the application sanitizes user input before passing to mail systems to protect against SMTP or IMAP injection.

5.2.4 Verify that the application avoids the use of eval() or other dynamic code execution features. Where there is no alternative, any user input being included must be sanitized or sandboxed before being executed.

5.2.4 Verify that the application avoids the use of eval() or other dynamic code execution features. Where there is no alternative, any user input being included must be sanitized or sandboxed before being executed.

5.2.6 Verify that the application protects against SSRF attacks, by validating or sanitizing untrusted data or HTTP file metadata, such as filenames and URL input fields, and uses allow lists of protocols, domains, paths and ports.

5.2.7 Verify that the application sanitizes, disables, or sandboxes user-supplied Scalable Vector Graphics (SVG) scriptable content, especially as they relate to XSS resulting from inline scripts, and foreignObject.

5.2.8 Verify that the application sanitizes, disables, or sandboxes user-supplied scriptable or expression template language content, such as Markdown, CSS or XSL stylesheets, BBCode, or similar.

V5.3 Output Encoding and Injection Prevention

7.2.9. V5.3 Output Encoding and Injection Prevention

5.3.1 Verify that output encoding is relevant for the interpreter and context required. For example, use encoders specifically for HTML values, HTML attributes, JavaScript, URL parameters, HTTP headers, SMTP, and others as the context requires, especially from untrusted inputs (e.g. names with Unicode or apostrophes, such as ねこ or O'Hara).

5.3.2 Verify that output encoding preserves the user's chosen character set and locale, such that any Unicode character point is valid and safely handled.

5.3.3 Verify that context-aware, preferably automated - or at worst, manual - output escaping protects against reflected, stored, and DOM based XSS.

5.3.4 Verify that data selection or database queries (e.g. SQL, HQL, ORM, NoSQL) use parameterized queries, ORMs, entity frameworks, or are otherwise protected from database injection attacks.

5.3.5 Verify that where parameterized or safer mechanisms are not present, context-specific output encoding is used to protect against injection attacks, such as the use of SQL escaping to protect against SQL injection.

5.3.6 Verify that the application protects against JSON injection attacks, JSON eval attacks, and JavaScript expression evaluation.

5.3.7 Verify that the application protects against LDAP injection vulnerabilities, or that specific security controls to prevent LDAP injection have been implemented.

5.3.8 Verify that the application protects against OS command injection and that operating system calls use parameterized OS queries or use contextual command line output encoding.

5.3.9 Verify that the application protects against Local File Inclusion (LFI) or Remote File Inclusion (RFI) attacks.

5.3.10 Verify that the application protects against XPath injection or XML injection attacks.

7.2.10. V5.4 Memory, String, and Unmanaged Code

5.4.1 Verify that the application uses memory-safe string, safer memory copy and pointer arithmetic to detect or prevent stack, buffer, or heap overflows.

5.4.2 Verify that format strings do not take potentially hostile input, and are constant.

5.4.3 Verify that sign, range, and input validation techniques are used to prevent integer overflows.

7.2.11. V5.5 Deserialization Prevention

5.5.1 Verify that serialized objects use integrity checks or are encrypted to prevent hostile object creation or data tampering.

5.5.2 Verify that the application correctly restricts XML parsers to only use the most restrictive configuration possible and to ensure that unsafe features such as resolving external entities are disabled to prevent XML eXternal Entity (XXE) attacks.

5.5.3 Verify that deserialization of untrusted data is avoided or is protected in both custom code and third-party libraries (such as JSON, XML and YAML parsers).

5.5.4 Verify that when parsing JSON in browsers or JavaScript-based backends, JSON.parse is used to parse the JSON document. Do not use eval() to parse JSON.

7.2.12. V6.2 Algorithms

6.2.1 Verify that all cryptographic modules fail securely, and errors are handled in a way that does not enable Padding Oracle attacks.V7.4 Error Handling

7.2.13. V7.4 Error Handling

7.4.1 Verify that a generic message is shown when an unexpected or security sensitive error occurs, potentially with a unique ID which support personnel can use to investigate.

7.4.2 Verify that exception handling (or a functional equivalent) is used across the codebase to account for expected and unexpected error conditions.

7.4.3 Verify that a "last resort" error handler is defined which will catch all unhandled exceptions.

7.2.14. V10.2 Malicious Code Search

10.2.1 Verify that the application source code and third party libraries do not contain unauthorized phone home or data collection capabilities. Where such functionality exists, obtain the user's permission for it to operate before collecting any data.

10.2.4 Verify that the application source code and third party libraries do not contain time bombs by searching for date and time related functions.

10.2.5 Verify that the application source code and third party libraries do not contain malicious code, such as salami attacks, logic bypasses, or logic bombs.

10.2.6 Verify that the application source code and third party libraries do not contain Easter eggs or any other potentially unwanted functionality.

7.2.15. V10.3 Application Integrity

10.3.2 Verify that the application employs integrity protections, such as code signing or subresource integrity. The application must not load or execute code from untrusted sources, such as loading includes, modules, plugins, code, or libraries from untrusted sources or the Internet.

10.3.3 Verify that the application has protection from subdomain takeovers if the application relies upon DNS entries or DNS subdomains, such as expired domain names, out of date DNS pointers or CNAMEs, expired projects at public source code repos, or transient cloud APIs, serverless functions, or storage buckets (autogen-bucket-id.cloud.example.com) or similar. Protections can include ensuring that DNS names used by applications are regularly checked for expiry or change.

7.2.16. V11.1 Business Logic Security

11.1.1 Verify that the application will only process business logic flows for the same user in sequential step order and without skipping steps.

11.1.2 Verify that the application will only process business logic flows with all steps being processed in realistic human time, i.e. transactions are not submitted too quickly.

11.1.3 Verify the application has appropriate limits for specific business actions or transactions which are correctly enforced on a per user basis.

11.1.4 Verify that the application has anti-automation controls to protect against excessive calls such as mass data exfiltration, business logic requests, file uploads or denial of service attacks.

11.1.5 Verify the application has business logic limits or validation to protect against likely business risks or threats, identified using threat modeling or similar methodologies.

11.1.6 Verify that the application does not suffer from "Time Of Check to Time Of Use" (TOCTOU) issues or other race conditions for sensitive operations.

11.1.7 Verify that the application monitors for unusual events or activity from a business logic perspective. For example, attempts to perform actions out of order or actions which a normal user would never attempt.

11.1.8 Verify that the application has configurable alerting when automated attacks or unusual activity is detected.

7.2.17. V12.1 File Upload

12.1.1 Verify that the application will not accept large files that could fill up storage or cause a denial of service.

12.1.2 Verify that the application checks compressed files (e.g. zip, gz, docx, odt) against maximum allowed uncompressed size and against maximum number of files before uncompressing the file.

12.1.3 Verify that a file size quota and maximum number of files per user is enforced to ensure that a single user cannot fill up the storage with too many files, or excessively large files.

12.2.1 Verify that files obtained from untrusted sources are validated to be of expected type based on the file's content.

7.2.18. V12.3 File Execution

12.3.1 Verify that user-submitted filename metadata is not used directly by system or framework filesystems and that a URL API is used to protect against path traversal.

12.3.2 Verify that user-submitted filename metadata is validated or ignored to prevent the disclosure, creation, updating or removal of local files (LFI).

12.3.3 Verify that user-submitted filename metadata is validated or ignored to prevent the disclosure or execution of remote files via Remote File Inclusion (RFI) or Server-side Request Forgery (SSRF) attacks.

12.3.4 Verify that the application protects against Reflective File Download (RFD) by validating or ignoring user-submitted filenames in a JSON, JSONP, or URL parameter, the response Content-Type header should be set to text/plain, and the Content-Disposition header should have a fixed filename.

12.3.5 Verify that untrusted file metadata is not used directly with system API or libraries, to protect against OS command injection.

12.3.6 Verify that the application does not include and execute functionality from untrusted sources, such as unverified content distribution networks, JavaScript libraries, node npm libraries, or server-side DLLs.

7.2.19. V12.4 File Storage

12.4.1 Verify that files obtained from untrusted sources are stored outside the web root, with limited permissions.

12.4.2 Verify that files obtained from untrusted sources are scanned by antivirus scanners to prevent upload and serving of known malicious content.

7.2.20. V13.1 Generic Web Service Security

13.1.1 Verify that all application components use the same encodings and parsers to avoid parsing attacks that exploit different URI or file parsing behavior that could be used in SSRF and RFI attacks.

13.1.3 Verify API URLs do not expose sensitive information, such as the API key, session tokens etc.

13.1.4 Verify that authorization decisions are made at both the URI, enforced by programmatic or declarative security at the controller or router, and at the resource level, enforced by model-based permissions.

13.1.5 Verify that requests containing unexpected or missing content types are rejected with appropriate headers (HTTP response status 406 Unacceptable or 415 Unsupported Media Type).

7.2.21. V14.1 Build and Deploy

14.1.1 Verify that the application build and deployment processes are performed in a secure and repeatable way, such as CI / CD automation, automated configuration management, and automated deployment scripts.

14.1.2 Verify that compiler flags are configured to enable all available buffer overflow protections and warnings, including stack randomization, data execution prevention, and to break the build if an unsafe pointer, memory, format string, integer, or string operations are found.

7.2.22. V14.3 Unintended Security Disclosure

14.3.2 Verify that web or application server and application framework debug modes are disabled in production to eliminate debug features, developer consoles, and unintended security disclosures.

8. MITRE

8.1. Application Developer Guidance

Link: https://attack.mitre.org/mitigations/M1013/

8.1.1. T1564.009 Hide Artifacts: Resource Forking

Configure applications to use the application bundle structure which leverages the /Resources folder location.

8.1.2. T1517 Access Notifications

Application developers could be encouraged to avoid placing sensitive data in notification text.

8.1.3. T1635 Steal Application Access Token

Developers should use Android App Links and iOS Universal Links to provide a secure binding between URIs and applications, preventing malicious applications from intercepting redirections. Additionally, for OAuth use cases, PKCE should be used to prevent use of stolen authorization codes.

8.1.4. T1635.001 URI Hijacking

Developers should use Android App Links and iOS Universal Links to provide a secure binding between URIs and applications, preventing malicious applications from intercepting redirections. Additionally, for OAuth use cases, PKCE should be used to prevent use of stolen authorization codes.

8.1.5. T1474 Supply Chain Compromise

Application developers should be cautious when selecting third-party libraries to integrate into their application.

8.1.6. T1474.001 Compromise Software Dependencies and Development Tools

Application developers should be cautious when selecting third-party libraries to integrate into their application.

9. UK National Cyber Security Centre (NCSC)

9.1. Application development Recommendations

Link: https://www.ncsc.gov.uk/collection/application-development/generic-application-development

9.1.1. Application hardening Stack protection

Compile native code portions of the application to take advantage of any protection mechanisms that are available on the platform. Enable features such as Address Space Layout Randomisation (ASLR) and Stack Canaries during compilation in order to make the application more difficult to exploit. However, these should only be used to increase the effort needed to exploit vulnerabilities, and should not be solely relied upon to prevent exploitation.

9.1.2. Application hardening Code obfuscation

You can take steps to make your applications more difficult to reverse engineer, but it's important that applications remain secure even when the entire system is understood by an attacker. As such, obfuscation techniques should only be considered to prevent the reverse engineering of technologies to safeguard (for example) intellectual property, and not to provide a robust security system. Even so, you can use obfuscation of both native and managed code to make reverse engineering attempts more difficult. This is also likely to increase the effort required for attackers to understand how to attack and break the application’s security model.

9.1.3. Application hardening Jailbreak and root detection

Jailbroken or rooted devices are a threat to sensitive data they contain. As a method of hardening the application, consider implementing checks to detect if the device has been compromised. Detection will always be subject to circumvention by a determined attacker, however tests for common jailbreak and rooting methods allow for the application to take appropriate steps, such as alerting the user, or preventing the device from processing sensitive information.

9.1.4. Android application development 1.3 Secure application development Application security

To hinder the exploitation of any potential memory corruption vulnerabilities, the following recommendations should be followed:
The application should be compiled using the latest supported compiler security flags.
The application should not be compiled with the debug flag enabled.
The application should not use any private APIs.
The application should be compiled in release mode with all debug information stripped from the binaries.
If Android Studio is used, it should be configured to shrink and optimise Java code.

9.1.5. Android application development 1.3 Secure application development Security recommendations

The behaviours listed below can increase the overall security of an application.

Any data that is deemed necessary to store on the device should be encrypted either with keys that are not stored on the device, or that are stored in the Android KeyStore. Furthermore, key attestation should be used when hardware-backed key storage is available.
Where possible, applications should sanitise in-memory buffers of sensitive data after use (if the data is no longer required for operation).
Applications that require authentication on application launch should also request this authentication credential when returning back to the foreground after previously being backgrounded by a user, allowing for a small grace period.
As the standard Android clipboard is shared between all applications on the device, do not not use it when accessing sensitive data. A private clipboard can be implemented if required by the application.
The application should disable both manual and automatic screenshots within activities that display sensitive data by setting secure flags of the window within the application.
Applications that use a shared UID will share the same sandbox. This means that if one application was compromised, all data in any application with a shared UID would also be compromised. Developers should share functionality between applications using intents, restricted by permissions.
Intents created for IPC between trusted applications should use signature permissions to restrict access by other applications on the device.
Applications that use Web Views should limit the features and capabilities to the minimum functionality required.
JavaScript and local file access should be disabled unless specifically required.
Caching should be disabled to prevent unnecessary exposure of sensitive data.
The application should ensure that debugging output has been removed and sensitive information prevented from appearing within the device log files.

9.1.6. Apple iOS application development

Recommendations for the secure development, procurement and deployment of Apple iOS applications.
This guidance contains recommendations for the secure development,procurement and deployment of iOS applications. Please familiarise yourself with the generic application development guidance section before continuing.

Note that this guidance also refers to concepts described in the NCSC End User Devices Security Guidance. We recommend you familiarise yourself with this before reading the guidance below.
A general guide to secure Apple programming can be found in Apple's security guide.
Regarding data at rest and keychain protection classes, the following terminology will be used:

Availability class name Data protection class Keychain protection class
A (when unlocked) NSFileProtectionComplete kSecAttrAccessibleWhenUnlocked
B (while unlocked) NSFileProtectionCompleteUnlessOpen N/A
C (after first unlock) NSFileProtectionCompleteUntilFirstUserAuthentication (default on iOS 7 and above) kSecAttrAccessibleAfterFirstUnlock
D (always) NSFileProtectionNone kSecAttrAccessibleAlways
Passcode enabled N/A kSecAttrAccessible WhenPasscodeSetThisDeviceOnly*
Note that the other keychain classes have a 'This device only' counterpart. More information about these protection classes can be found within Apple's security guide document and API documentation.

9.1.7. Secure iOS application development 1.3 Secure application development recommendations Secure data transmission

In order to transmit sensitive data securely, iOS applications should conform to the following:

All off-device communications must take place over a mutually-authenticated, cryptographically-protected connection. For example:
the assured IPsec VPN to the corporate network
Secure Chorus for secure real-time media streaming such as secure voice
Transport Layer Security (TLS) with certificate pinning to a known endpoint to a service within the corporate network; more information on TLS can be found within NCSC's TLS documentation
The application must not allow its sensitive data to be opened in other applications on the device (e.g. through Open In) unless that application is on an appropriate enterprise-managed allow list.
Any security-critical settings (such as server addresses and certificates) must be defined at build time or be enterprise-managed. The user must not be able to alter these settings.
If cloud services, such as iCloud, are used by the app to store information, this should be protected using the appropriate encryption mechanisms both from a network transmission perspective and data storage perspective.
Applications should make use of App Transport Security (ATS) and should not disable this feature or add domains to the exception allow list.
Applications should aim to keep perfect forward secrecy (PFS) enabled and not reduce the minimum TLS version supported.
Ensure that kSecAttrSynchronizable is not set for security-sensitive keychain items (as they will be included in an iCloud keychain backup, if this functionality is enabled).
Note that at present there is no API on iOS to check the status of the VPN. To securely check the status of the VPN, the internal service with which the application is communicating must be authenticated. The recommended way of performing this authentication is TLS with a pinned certificate. If mutual authentication is required to the internal service, mutual TLS with pinned certificates should be used.

9.1.8. Secure iOS application development 1.3 Secure application development recommendations Application security

To hinder the exploitation of any potential memory corruption vulnerabilities, the following recommendations should be followed:

The application should be compiled using the latest supported security flags.
The binary should be compiled with the Process Independent Executable (PIE) flag set.
The application should make use of Automatic Reference Counting (ARC) – which is enabled by default.
The application should not use private APIs.

9.1.9. Secure iOS application development 1.3 Secure application development recommendations Client side security

The following recommendations should be followed to improve the security of the client:

Secure coding practices should be followed to protect against input injection attacks. More information can be found in Apple's secure coding guide.
If the application uses Web Views (UIWebView, WKWebView or FSafariViewController) it should disable features which are not required by content loaded into the WebView (for example JavaScript or local file access). This will lead to a reduction in attack surface and help protect this area of the application.
If content is being loaded locally into a Web View, users should be prevented from changing the filename or path which is loaded and they should not be able to edit the loaded file.

9.1.10. Secure Windows application development 1.7 Application security

In order to prevent potential memory corruption vulnerabilities and protect against reverse engineering, Windows applications should conform to the following:

The application should be compiled using the latest supported security flags.
The application should be compiled in release mode with all debug information stripped from the binaries.
When applications are updated, the new version should target the latest SDK version.

10. National Information Assurance Partnership (NIAP)

10.1. Requirements for Vetting Mobile Apps from the Protection Profile for Application Software

Link: https://www.niap-ccevs.org/MMO/PP/394.R/pp_app_v1.2_table-reqs.htm

10.1.1. Anti-Exploitation Capabilities FPT_AEX_EXT.1.3

The application shall be compatible with security features provided by the platform vendor.
Application Note: This requirement is designed to ensure that platform security features do not need to be disabled in order for the application to run.

10.1.2. Anti-Exploitation Capabilities FPT_AEX_EXT.1.5

The application shall be compiled with stack-based buffer overflow protection enabled.

10.1.3. Integrity for Installation and Update FPT_TUD_EXT.1.3

The application shall be packaged such that its removal results in the deletion of all traces of the application, with the exception of configuration settings, output files, and audit/log events.
Application Note: Applications bundled with the system/firmware image are not subject to this requirement if the user is unable to remove the application through means provided by the OS.

10.1.4. Integrity for Installation and Update FPT_TUD_EXT.1.4

The application shall not download, modify, replace or update its own binary code.
Application Note: This requirement applies to the code of the application; it does not apply to mobile code technologies that are designed for download and execution by the application.

10.1.5. Security Assurance Requirements ALC_CMC.1.1C

The application shall be labeled with a unique reference.
Application Note: Unique reference information includes:
Application Name
Application Version
Application Description
Platform on which Application Runs
Software Identification (SWID) tags, if available

10.1.6. Use of Supported Services and APIs FPT_API_EXT.2.1

The application [selection: shall use platform-provided libraries, does not implement functionality] for parsing [assignment: list of formats parsed that are included in the IANA MIME media types] .
Application Note: The IANA MIME types are listed at http://www.iana.org/assignments/media-types and include many image, audio, video, and content file formats. This requirement does not apply if providing parsing services is the purpose of the application.

10.1.7. Software Identification and Versions FPT_IDV_EXT.1.1

The application shall include SWID tags that comply with the minimum requirements for SWID tag from ISO/IEC 19770-2:2009 standard. Application Note: Valid SWID tags must contain a Software Identity element and an Entity element as defined in the ISO/IEC 19770-2:2009 standard. SWID tags must be stored with a .swidtag file extensions as defined in the ISO/IEC 19770-2:2009.