Welcome to ACTk

Thank you for choosing the **Anti-Cheat Toolkit (ACTk)** for Unity. This manual walks you through the toolkit's capabilities, recommended workflows, and practical tips that help you ship a resilient player experience.

> **NOTE**
>
> If you need scripting reference details, jump straight to the **API documentation**. This manual focuses on usage patterns, configuration, and decision-making guidance.

How to use this manual

The user manual is organized into modular topics so you can quickly jump to the material you need:

- Getting started – install the package, configure global settings, and review the quick-start checklist.
- Obscured data types – fight memory editors, serialize data safely, and understand DOTS Hybrid workflows.
- Secure storage – protect save files and preferences, manage device locking, and use the Prefs Editor effectively.
- Detectors – configure cheat detectors, tune thresholds, and react to cheating attempts in code or the editor.
- Code integrity & platform tooling – validate build integrity, harden your delivery pipeline, and leverage platform-specific helpers.
- Integrations & ecosystem – review official add-ons and third-party solutions that extend ACTk.
- Troubleshooting – diagnose common pitfalls and keep your project stable.
- Migration – update existing projects safely when upgrading from older ACTk releases.
- Compatibility – review supported platforms, export compliance, and build tool notes.
- Support & resources – stay in touch, get help, and keep your team informed.

Each topic includes **best practice checklists** and **cross-links** so you can design a protection strategy that fits your game.

Quick reference

- **Homepage**
- **Community Discord**
- **Video tutorials**
- **Forum thread**

Disclaimer

Anti-cheat solutions can never be 100% unbreakable—skilled and motivated attackers can eventually bypass any client-side protection. The goal of ACTk is to stop the majority of cheaters, raise the cost of an attack, and provide you with telemetry you can act on.

Anti-cheat tooling should be part of a layered security strategy. Combine ACTk with authoritative server logic, code obfuscation, and your own monitoring. Keep ACTk updated—new releases ship fixes for newly discovered attack vectors and compatibility changes.

When you plan feature work, prefer **IL2CPP** builds where possible and make a habit of testing with the sample scenes in the package. Hands-on testing is still the best way to validate your protection setup.

Acknowledgments

Huge thanks to Daniele Giardini (Demigiant) for the awesome ACTk logos, intensive help, and priceless feedback.

Getting started with ACTk

Install and upgrade safely

1. Remove any previous ACTk version from your project before importing a new package release. This avoids leftover files that could conflict with the updated structure.
2. Import the package. Unity adds menu entries under `Tools > Code Stage > Anti-Cheat Toolkit` and `GameObject > Create Other > Code Stage > Anti-Cheat Toolkit`.
3. Open the sample scenes under `CodeStage/AntiCheatToolkit/Examples` and try to cheat them. First-hand experience is invaluable when you plan your own counter-measures.

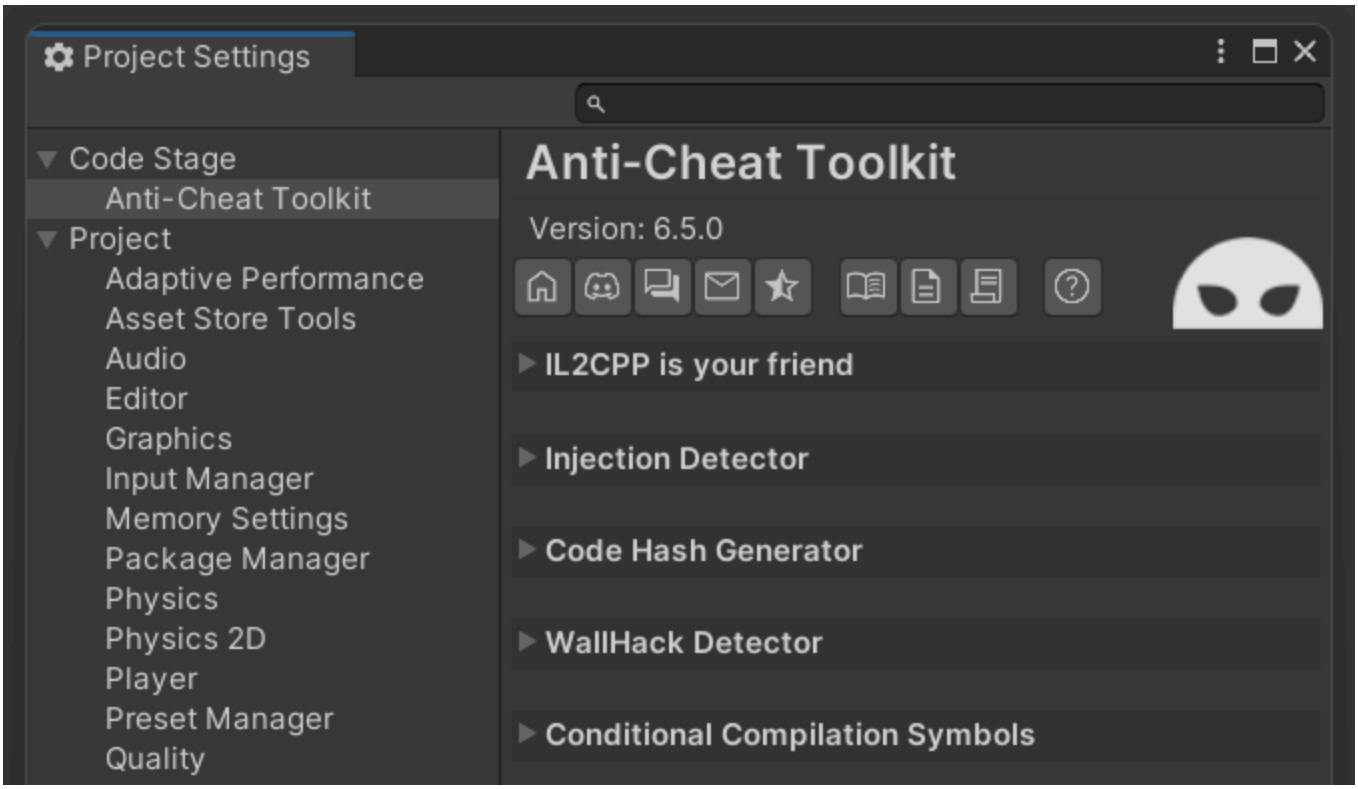
> **WARNING**
>
> Upgrades can modify serialization backends—back up your project or commit changes before importing a new package.

Available example scenes

- **API Examples**: `Examples/API Examples/API Examples.unity` - Comprehensive examples of all ACTk features
- **Code Genuine Validation**: `Examples/Code Genuine Validation/GenuineValidator.unity` - Advanced code hash validation
- **DOTS ECS Examples**: `Examples/DOTS ECS Examples/ACTk Hybrid DOTS Setup.unity` - Unity DOTS integration

The ACTk Settings window

Launch `Tools > Code Stage > Anti-Cheat Toolkit > Settings` to configure global options. Settings are stored in `ProjectSettings/ACTkSettings.asset`.



Key sections:

- **Toolkit status** – track the installed version and access quick links (support, Discord, this manual, the API reference, etc.).
- **IL2CPP is your friend** – highlights why IL2CPP builds are harder to reverse engineer and lets you switch scripting backends when supported.
- **Detectors** – enable automatic link.xml generation, adjust backlogs, and toggle debug logging.
- **Code Hash Generator** – turn on automatic hash generation after builds and configure whitelist behavior.
- **Conditional compilation symbols** – enable platform-specific helpers and debug switches (e.g., `ACTK_DETECTION_BACKLOGS`, `ACTK_INJECTION_DEBUG`). Common symbols include:
  - `ACTK_DETECTION_BACKLOGS` – print detector histories in dev builds
  - `ACTK_INJECTION_DEBUG`, `ACTK_INJECTION_DEBUG_VERBOSE`, `ACTK_INJECTION_DEBUG_PARANOID` – injection diagnostics levels
  - `ACTK_WALLHACK_DEBUG` – keep WH helpers visible and log more
  - `ACTK_WALLHACK_LINK_XML` – auto-generate `link.xml` for WH on IL2CPP
  - `ACTK_PREVENT_INTERNET_PERMISSION` – disable Time Cheating Detector and avoid INTERNET permission
  - `ACTK_PREVENT_READ_PHONE_STATE` – avoid READ_PHONE_STATE; provide custom `DeviceIdHolder.DeviceId` if you still need Device Lock
  - `ACTK_EXCLUDE_OBFUSCATION` – adds `[Obfuscation(Exclude = true)]` on sensitive methods to avoid obfuscator side effects
  - `ACTK_DEV_LOGS` – enable generic development logs in the Editor and development builds
  - `ACTK_US_EXPORT_COMPATIBLE` – downgrade crypto for export compliance (reduced security)

- `ACTK_NEWTONSOFT_JSON` – enable Json.NET converter for obscured types
- `ACTK_IS_HERE` – optional marker for third-party integrations

Quick-start checklist

Essential steps to get ACTk working in your project:

- [ ] **Import the package** and explore the sample scenes
- [ ] **Switch to IL2CPP** where possible for better protection
- [ ] **Pick tools for your game** - choose from ObscuredTypes, Detectors, Secure Storage, Code Integrity and other features
- [ ] **Configure settings** in the ACTk Settings window
- [ ] **Test your protection** in both Editor and builds

Editor tooling overview

The `Tools > Code Stage > Anti-Cheat Toolkit` menu provides shortcuts to:

- **Settings** – configure global options and conditional symbols.
- **Prefs Editor** – inspect and edit `PlayerPrefs` and `ObscuredPrefs` in a dedicated window.
- **Injection Detector Whitelist Editor** – maintain assemblies that are safe to load at runtime.
- **Code hash utilities** – calculate hashes for existing builds and automate proguard configuration.
- **Migration and Validation** - run migration or validation utilities to help you update from older ACTk versions.

Protecting memory with obscured types

Cheaters frequently rely on memory editors (Cheat Engine, Game Guardian, etc.) to locate and modify variables such as health, currency, and timers. ACTk counters these tools with a family of **obscured types** that transparently encrypt values, randomize keys, and expose hooks for detecting tampering.

```
private ObscuredInt coins = 100;
public void Add(int amount) => coins += amount;
```

Obscured types behave exactly like their native counterparts, but store encrypted values in memory. Casting between native and obscured types is implicit.

Feature highlights

- Drop-in replacements for the most common C# and Unity struct types (`ObscuredInt`, `ObscuredFloat`, `ObscuredVector3`, etc.).
- Transparent encryption with randomized keys and support for honeypots through the Obscured Cheating Detector.
- Serialization helpers for Json.NET and JsonUtility.
- APIs to access or replace encrypted payloads when you integrate custom persistence layers.

Usage guidelines

- Place the `using CodeStage.AntiCheat.ObscuredTypes;` directive at the top of each file that needs obscured types.
- Replace only vulnerable members that are typically long-living fields, properties, and value sources. There's usually no need to use obscured types as method arguments or local variables if they don't live longer than a few frames, due to the nature of the memory hacking process—cheaters need multiple value searches to filter out target values from many addresses, which can't be completed in one frame.
- Large arrays or hot update loops should remain regular types to avoid performance penalties.
- ⚠️ **Data Loss Warning**: When migrating inspector-exposed fields to obscured types, expect serialized values to reset completely. This can lead to permanent data loss if not planned carefully. Always backup your project and plan migrations accordingly.
- Prefer storing obscured values outside `Update` loops, and profile on mobile targets to catch hotspots early.

> **WARNING**
>
> Enable the **Obscured Cheating Detector** to catch cheat attempts even without honeypots. Each obscured type maintains internal integrity checks, so any tampering with encrypted values will trigger detection. Honeypots are optional and provide additional protection by serving fake unencrypted values—attackers who freeze these honeypot values will also trigger detection events you can react to.

Serialization support

Json.NET

If your project uses Newtonsoft Json.NET, you can use the bundled `ObscuredTypesNewtonsoftConverter` for serializing and deserializing ObscuredTypes decrypted values.

For detailed usage examples and API reference, see ObscuredTypesNewtonsoftConverter API documentation.

> **NOTE**
>
> The converter is wrapped in the `ACTK_NEWTONSOFT_JSON` symbol. Unity automatically defines it when the `com.unity.nuget.newtonsoft-json` package (v2.0.0+) is installed. Add the symbol manually if you ship a custom Json.NET build, or use the ACTk Settings (Conditional Compilation Symbols) to manage symbols.

Unity JsonUtility

JsonUtility is less flexible but still workable. Implement ISerializationCallbackReceiver and move encrypted data into regular fields inside `OnBeforeSerialize` / `OnAfterDeserialize`. The forum example demonstrates the pattern.

Additional serialization considerations

- **LINQ and XmlSerializer** are not supported at this moment.
- **Binary serialization** is supported out of the box, but be careful with `BinaryFormatter` as it's considered non-safe.
- **ISerializable alternative**: As an alternative to the JsonConverter, it's possible to implement `ISerializable` as shown in this example.
- **General recommendation**: Cast obscured variables to regular ones for advanced operations, then cast back to obscured types to ensure compatibility.

Advanced usage

- **Custom encryption keys**: Use `GenerateKey()`, `Encrypt()`, and `Decrypt()` for custom key management
- **Raw encrypted data**: Access `GetEncrypted()` and `SetEncrypted()` for custom serialization
- **Key randomization**: Call `RandomizeCryptoKey()` periodically to invalidate memory searches

For detailed API reference and code examples, see ObscuredInt API documentation. Other obscured types share the similar logic and approaches.

DOTS Hybrid workflows

ACTk provides official DOTS Hybrid support for ECS-based games:

Key Points

- `ObscuredFloat`, `ObscuredDouble`, `ObscuredInt`, `ObscuredLong`, `ObscuredBool`, `ObscuredVector3`, and most other obscured types can be used in `IComponentData` implementations.
- Detectors and other native-specific features work fine in Hybrid mode.
- The package ships an end-to-end sample under `CodeStage/AntiCheatToolkit/Examples/DOTS ECS Examples`.

Installation

1. Install the required DOTS packages: `com.unity.entities`, `com.unity.rendering.hybrid`, and `com.unity.dots.editor`.
2. Open **ACTk Hybrid DOTS Setup** scene and enter Play mode.
3. Use the in-game HUD to modify obscured values and trigger the DOTS-compatible detectors.

> **CAUTION**
>
> Reference types such as `ObscuredString`, ECS non-compatible types such as `ObscuredDecimal`, and other managed-only types remain incompatible with `IComponentData` and should not be stored inside Entities. Keep them on authoring MonoBehaviours instead, here is an example.

Diagnostics and validation

- Add the **Obscured Cheating Detector** to ensure modified encrypted values trigger a callback.
- Use the **API Examples** scene to profile performance and try edge cases.
- Use the validation tools from the menu (`Tools > Code Stage > Anti-Cheat Toolkit > Migrate > ...`) if you suspect serialized instances became corrupted after a package upgrade.

Securing saves and preferences

Protecting local storage is crucial when your game relies on client-side data for progression, in-app purchases, or cooldown timers. ACTk ships with three complementary systems—**ObscuredFile**, **ObscuredFilePrefs**, and **ObscuredPrefs**—plus editor tooling to inspect data during development.

Choosing the right storage solution

Feature	ObscuredPrefs	ObscuredFilePrefs	ObscuredFile
Backend	Unity PlayerPrefs	File system	File system
API Style	PlayerPrefs-like	PlayerPrefs-like	Raw byte arrays
Performance	PlayerPrefs + Binary Serialization + Encryption	File I/O + Binary Serialization + Encryption	File I/O + Encryption
Data Size	Small values only	Moderate size (caches data in memory)	Any size (no cache, direct I/O)
Thread Safety	Main thread only	Background thread safe	Background thread safe
Auto-save	Automatic	Automatic (configurable)	Manual
Use Cases	Settings, small progress data	Moderate save files, complex data	Large save files and raw data

ObscuredFile

`ObscuredFile` provides an encrypted file wrapper that detects tampering and optionally locks data to a specific device or custom identifier.

Basic Usage

```
var file = new ObscuredFile();
var result = file.WriteAllBytes(data);
if (result.Success) Debug.Log("Saved!");
```

Advanced Features

- **Device lock**: Use `DeviceLockSettings` to bind data to specific devices
- **Custom encryption**: Configure `EncryptionSettings` with custom passwords
- **Async operations**: Call `UnityApiResultsHolder.InitForAsyncUsage(true)` for background operations

- **Event handling**: Subscribe to `DataFromAnotherDeviceDetected` and `NotGenuineDataDetected`

For detailed API reference and examples, see ObscuredFile API documentation.

Key Capabilities

- Encrypts payloads (optional) to hide sensitive information from plain-text inspection.
- Validates integrity even when you disable encryption—cheaters cannot silently edit stored values.
- Locks the file to a device ID or your own identifier. See Device lock for configuration tips.
- Works with raw byte arrays so you can integrate any serialization stack.

> **WARNING**
>
> Always call `UnityApiResultsHolder.InitForAsyncUsage(true);` from the main thread before interacting with `ObscuredFile` from background threads.

ObscuredFilePrefs

`ObscuredFilePrefs` builds on top of `ObscuredFile` and emulates the familiar PlayerPrefs API while automatically encrypting values:

Basic Usage

```
ObscuredFilePrefs.Init();
ObscuredFilePrefs.Set("coins", 100);
var coins = ObscuredFilePrefs.Get("coins", 0);
```

Advanced Features

- **Custom settings**: Initialize with custom file name, encryption, and device lock settings
- **Data types**: Supports all Unity serializable types including Vector3, Color, DateTime, byte arrays
- **Async operations**: Use `UnityApiResultsHolder.InitForAsyncUsage(true)` for background operations
- **Key management**: `HasKey()`, `GetKeys()`, `DeleteKey()`, `DeleteAll()`
- **Memory management**: `LoadPrefs()`, `Save()`, `UnloadPrefs()` for memory control

For detailed API reference and examples, see ObscuredFilePrefs API documentation.

Example implementations

- **Basic usage**: See `ObscuredFilePrefsExamples.cs` in `Examples/API Examples/Scripts/Runtime/UsageExamples/`
- **DOTS integration**: See `UIActionSystem.cs` in `Examples/DOTS ECS Examples/Scripts/UI/Systems/`

Advantages

- Supports all basic C# types, `BigInteger`, byte arrays, `DateTime`, and common Unity structs.
- Emits events when data is tampered with or loaded from another device.
- AutoSave feature enabled by default to prevent data loss on app quit (desktop) and app losing focus (mobile).

> **TIP**
>
> Mix regular PlayerPrefs and ObscuredFilePrefs judiciously. Encrypt only the values that matter to control performance overhead.

ObscuredPrefs

`ObscuredPrefs` provides a direct replacement for Unity's `PlayerPrefs` with built-in encryption and tamper detection:

```
using CodeStage.AntiCheat.Storage;

ObscuredPrefs.Set("currentLifeBarObscured", 88.4f);
var currentLifeBar = ObscuredPrefs.Get("currentLifeBarObscured", 0f);

// same as
ObscuredPrefs.Set<float>("currentLifeBarObscured", 88.4f);
var currentLifeBar = ObscuredPrefs.Get<float>("currentLifeBarObscured");
```

Key features:

- **Drop-in replacement** for PlayerPrefs with automatic encryption
- **Extended type support** including all basic C# types, `BigInteger`, `byte[]`, `DateTime`, and Unity types
- **Device Lock support** to prevent save sharing between devices
- **Migration tools** to convert existing PlayerPrefs data automatically
- **Event system** with `NotGenuineDataDetected` for tamper detection

For detailed API reference and examples, see ObscuredPrefs API documentation.

Example implementations

- **Basic usage**: See `ObscuredPrefsExamples.cs` in `Examples/API Examples/Scripts/Runtime/UsageExamples/`
- **DOTS integration**: See `UIActionSystem.cs` in `Examples/DOTS ECS Examples/Scripts/UI/Systems/`

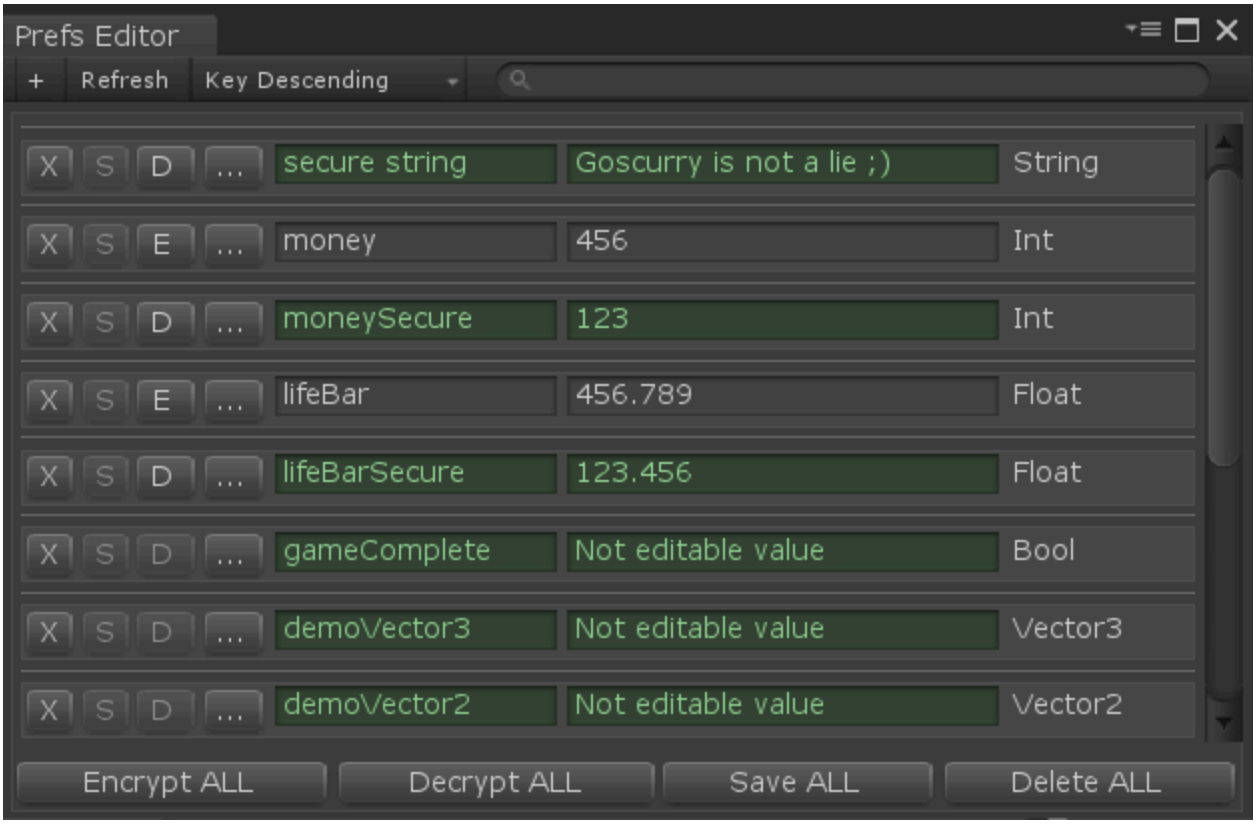
Migration tips:

- Replace `PlayerPrefs` calls with `ObscuredPrefs` throughout your project
- Use `preservePlayerPrefs` flag to keep original PlayerPrefs keys during migration
- Mix regular PlayerPrefs and ObscuredPrefs for different data types (use different key names)
- Use `ObscuredPrefs.DeleteAll()` instead of `PlayerPrefs.DeleteAll()` to properly clear internals

> **WARNING**
> ObscuredPrefs is slower than regular PlayerPrefs due to encryption overhead. Use it for sensitive data only, not for large datasets like maps or databases.

Prefs Editor

Use `Tools > Code Stage > Anti-Cheat Toolkit > Prefs Editor` to inspect and edit both PlayerPrefs and ObscuredPrefs directly in the editor.



Highlights:

- Add, edit, encrypt, and delete preferences without writing code
- Filter and sort large datasets (50 records per page) with pagination support
- Copy values to the clipboard and monitor progress when parsing large preference sets
- Works on Windows, macOS, and Linux editors

- Supports both PlayerPrefs and ObscuredPrefs in a unified interface
- Overwrite confirmation and progress bars for large datasets (1000+ records)
- Ignores Unity's internal prefs (UnitySelectMonitor, UnityGraphicsQuality, etc.)

> **NOTE**
> On Windows, prefs stored in Editor and standalone player are in different locations, so the Prefs Editor only works with Editor-saved preferences. Windows PlayerPrefs are stored in the registry at `HKEY_CURRENT_USER\Software\[Your Company]\[Your Product]`.

Device lock

`ObscuredFile`, `ObscuredFilePrefs`, and `ObscuredPrefs` can restrict data to a device or custom ID via `DeviceLockSettings`. This prevents save games from being shared between devices.

Lock levels

- **`None`** – data remains unlocked but can read both locked and unlocked data
- **`Soft`** – existing data stays readable; all new saves lock to the current device
- **`Strict`** – only accepts data locked to the current device; all new saves lock to the current device

Event handling

Subscribe to `DataFromAnotherDeviceDetected` to intercept foreign payloads:

- In `ObscuredPrefs`: fires once per session
- In `ObscuredFile`/`ObscuredFilePrefs`: fires every time a file is read

Platform considerations

> **WARNING**
> iOS does not provide a reliable persistent device identifier. If you rely on Device Lock, set a stable, app-defined identifier via `DeviceIdHolder.DeviceId` (for example, an authenticated account/user ID). Plan recovery flows using `DeviceLockTamperingSensitivity` when an identifier changes (SIM swap, device restore, reinstall). See related notes in Troubleshooting.

Android: Device Lock may require additional permissions depending on your setup. See Permissions and compliance for symbols to avoid unnecessary permissions in builds where Device Lock or Time Cheating Detector are not used.

Performance optimization

- Call `DeviceIdHolder.ForceLockToDeviceInit()` during loading screens to avoid CPU spikes on first device ID access
- Use `DeviceIdHolder.DeviceId` to set custom identifiers (useful for server-side account systems)

Sensitivity control

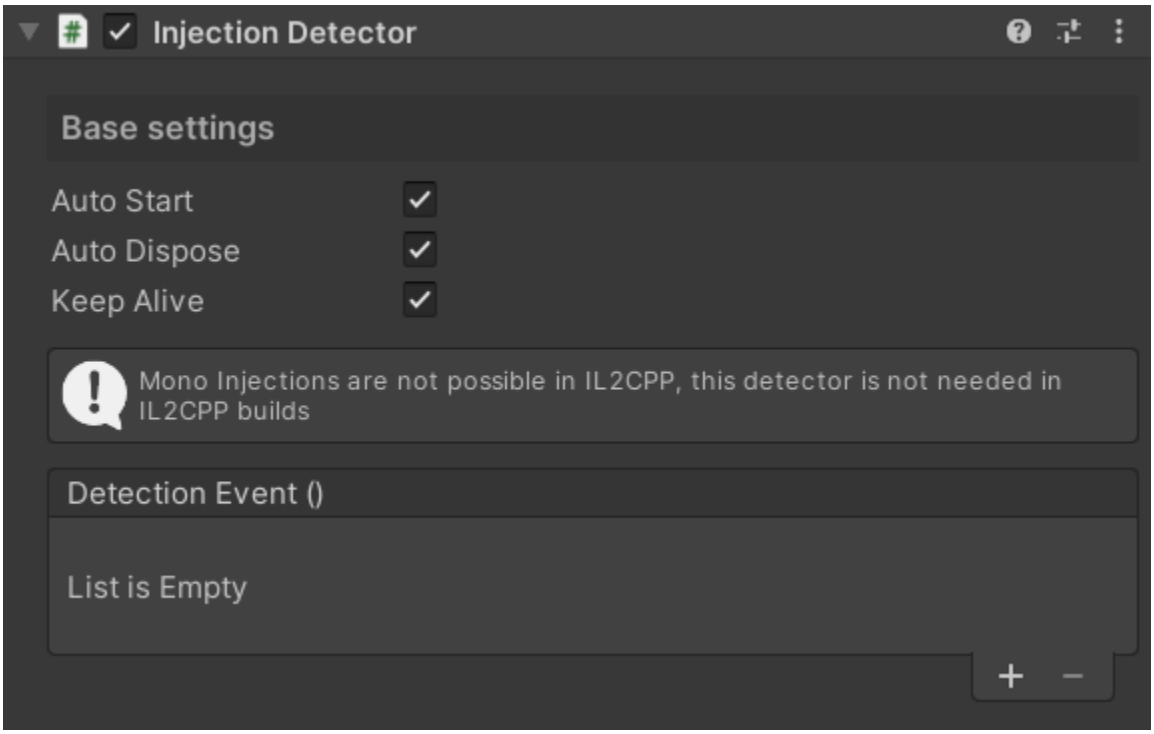
When using `Strict` mode, you can adjust `DeviceLockTamperingSensitivity`:

- `Normal` – default strict behavior, rejects foreign data and fires detection event
- `Low` – detection event still fires but data remains readable
- `Disabled` – no event fired, useful for data recovery when Device ID changes unexpectedly

Cheat detectors and responses

ACTk detectors continuously monitor your game for suspicious activity and raise callbacks you can use to react—log analytics, notify your backend, or block the player. Each detector can be configured in the editor or started from code.

Common configuration



Add detectors via `GameObject > Create Other > Code Stage > Anti-Cheat Toolkit` or attach them to existing objects through `Component > Code Stage > Anti-Cheat Toolkit`. Every detector exposes the same core options:

- **Auto Start** – automatically begins detection after the scene loads. Disable if you want to control the start timing from code.
- **Auto Dispose** – destroys the component after a detection event. Leave enabled when you only need a single detection per session.
- **Keep Alive** – persists through scene loads.
- **Detection Event** – UnityEvent invoked on detection. Hook this up to your response logic or use the scripting API.

Scripting API

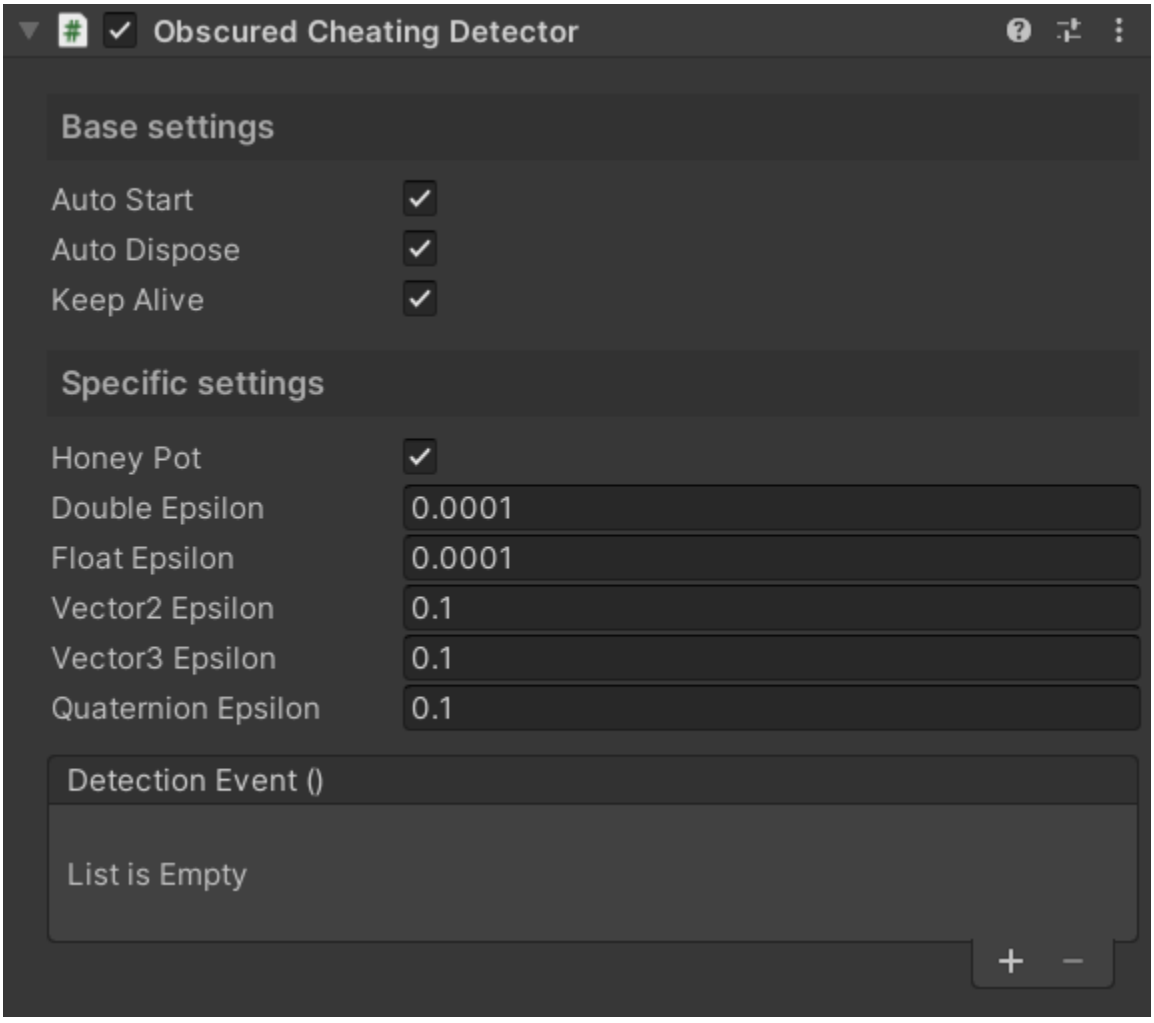
Control detectors programmatically via static methods:

```
SpeedHackDetector.StartDetection(OnSpeedHackDetected);
// ... handle detection
SpeedHackDetector.StopDetection();
```

Mix-and-match editor configuration with runtime control.

> **WARNING**
>
> Always start detectors from `Start()` or later lifecycle phases. Starting them in `Awake()` or `OnEnable()` can lead to race conditions when Unity initializes subsystems.

Obscured Cheating Detector



Catches tampering with obscured types by validating data and optionally presenting honeypot values while the encrypted payload stays safe.

Settings

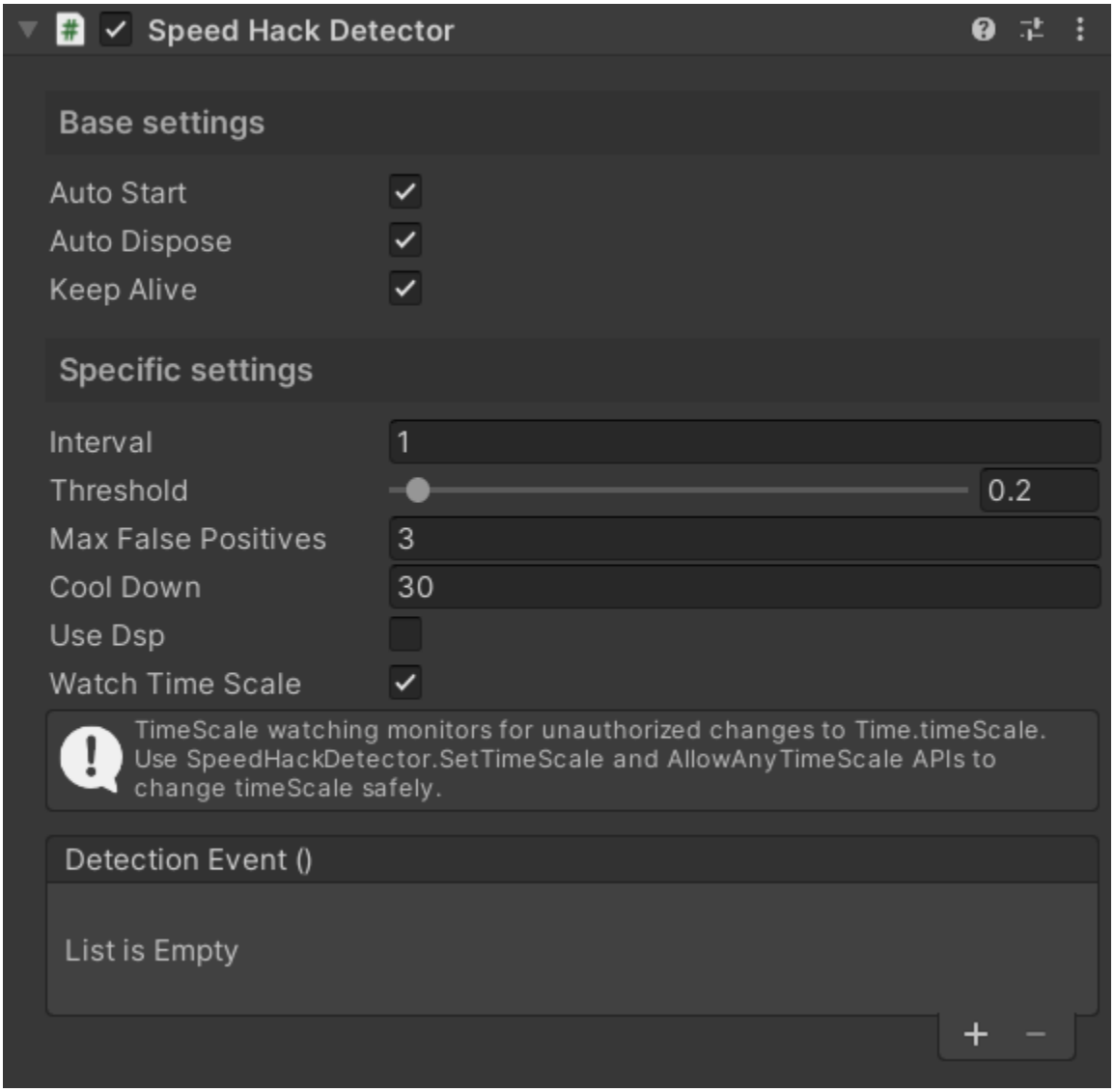
- **Honey Pot** – Enables fake unencrypted fields for attackers to find. Leave enabled to detect casual cheating attempts with easily discoverable fake values, disable if you only care about obfuscation and integrity validation.

- **Epsilons** – Tolerance thresholds for comparing fake and real values. Increase them if you see false positives with rapidly changing numbers:

    - **Double Epsilon** – Max allowed difference for ObscuredDouble values
    - **Float Epsilon** – Max allowed difference for ObscuredFloat values
    - **Vector2 Epsilon** – Max allowed difference for ObscuredVector2 values
    - **Vector3 Epsilon** – Max allowed difference for ObscuredVector3 values
    - **Quaternion Epsilon** – Max allowed difference for ObscuredQuaternion values

Works hand-in-hand with the Obscured types APIs.

Speed Hack Detector



Monitors discrepancies in timings to catch process acceleration / slowdown tools.

Basic Usage

```
SpeedHackDetector.StartDetection(OnSpeedHackDetected);
```

Settings

Core Detection Settings

- **Interval** – Time (in seconds) between detector checks. Lower values provide faster detection but may increase false positives on slower hardware.
- **Threshold** – Allowed speed multiplier threshold. Default: 0.2. Do not set too low values (e.g., 0 or 0.00*) since there are timer fluctuations on different hardware.
- **Max False Positives** – Maximum false positives count allowed before registering speed hack. Higher values reduce false positives but delay detection.
- **Cool Down** – Amount of sequential successful checks before clearing internal false positives counter. Set to 0 to disable. Higher values provide more stability but slower recovery from false positives.

Advanced Settings

- **Use DSP** – Controls whether to use DSP Timer to catch speed hacks in sandboxed environments (like WebGL, VMs, etc.). Uses AudioSettings.dspTime under the hood, which can catch some extra speed hacks but can potentially cause false positives on some hardware due to high sensitivity. ⚠️ **Warning:** Use at your peril!
- **Watch TimeScale** – Controls whether to watch Time.timeScale for unauthorized changes. When enabled, the detector will monitor for unauthorized changes to Time.timeScale. Use SetTimeScale() method to safely change timeScale without triggering false positives. ⚠️ **Warning:** May cause false positives if you change timeScale directly without using the provided API.

Advanced Features

SpeedHackProofTime APIs

When SpeedHackDetector is running, you can utilize `SpeedHackProofTime` class to use reliable timers instead of Unity's `Time.*` timers, which usually suffer from speed hacks. It mirrors most Unity timer properties except the fixed timestep values.

> **NOTE**
>
> SpeedHackProofTime may not work in some cases, e.g., when there is no legal way to reach reliable timers (may happen in sandboxed processes), it will fallback to usual timers instead.

Safe TimeScale Management

Use `SpeedHackDetector.SetTimeScale()` instead of directly setting `Time.timeScale` to avoid false positives. Use `AllowAnyTimeScaleFor()` for third-party assets that need to change timeScale temporarily.

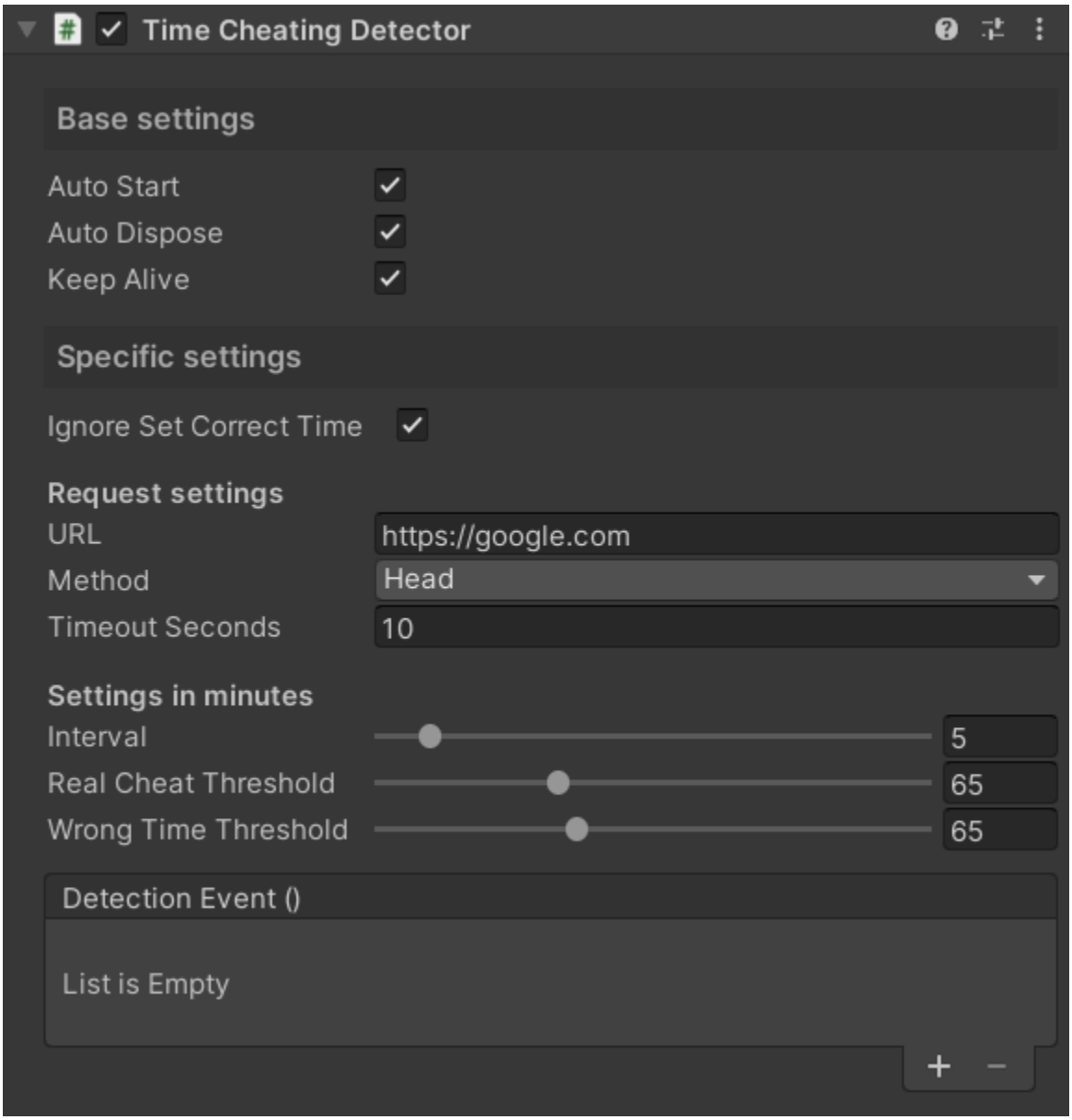
Configuration Guidelines

- Configure **Interval**, **Max False Positives**, and **Cool Down** to balance responsiveness and noise. For example: interval 1s, max false positives 5, cool down 30s.
- Consider using `SpeedHackProofTime` for critical timing operations that need to be cheat-proof.

> **NOTE**
>
> Reliable timers may be unavailable in sandboxed environments such as browser WebGL processes. Provide fallbacks or degrade gracefully.

For detailed examples and API reference, see SpeedHackDetector API documentation.

Time Cheating Detector



Detects changes to the system clock by comparing it to an authoritative online source.

Basic Usage

```
TimeCheatingDetector.StartDetection(OnTimeCheatingDetected);
```

Settings

Request Settings

- **Request URL** – Absolute URL which will return correct Date in response header to the HEAD request. Nearly any popular web server works including google.com, microsoft.com, etc.

- **Request Method** – Method to use for URL request. Options: Head (preferred, faster), Get (more compatible). Use Head if possible and fall back to Get if server blocks head requests.
- **Timeout Seconds** – Online time request timeout in seconds. Request will be automatically aborted if server doesn't respond in specified time.

Detection Settings

- **Interval** – Time (in **minutes**) between detector checks. Set to 0 to disable automatic time checks and use manual ForceCheck methods.
- **Real Cheat Threshold** – Threshold for detecting real time cheats (in **minutes**). If the difference between local and online time exceeds this value, it's considered a real cheat.
- **Wrong Time Threshold** – Threshold for detecting wrong time (in **minutes**). If the difference between local and online time exceeds this value but is less than real cheat threshold, it's considered wrong time.
- **Ignore Set Correct Time** – When enabled, ignores cases where time changes to be in sync with online correct time. Wrong time threshold is still taken into account. Helps reduce false positives when users correct their system time.

Advanced Features

Manual Time Checks

Use `ForceCheckAwaitable()` and `GetOnlineTimeAwaitable()` for on-demand validation.

For older Unity versions (pre-2023.1), use `ForceCheckEnumerator()` or `ForceCheckTask()` instead.

Custom Server Configuration

Configure custom time servers and thresholds using `RequestUrl` property or `StartDetection()` overloads.

Error Handling

Handle different `CheckResult` and `ErrorKind` values in your callback to manage network issues and detection results.

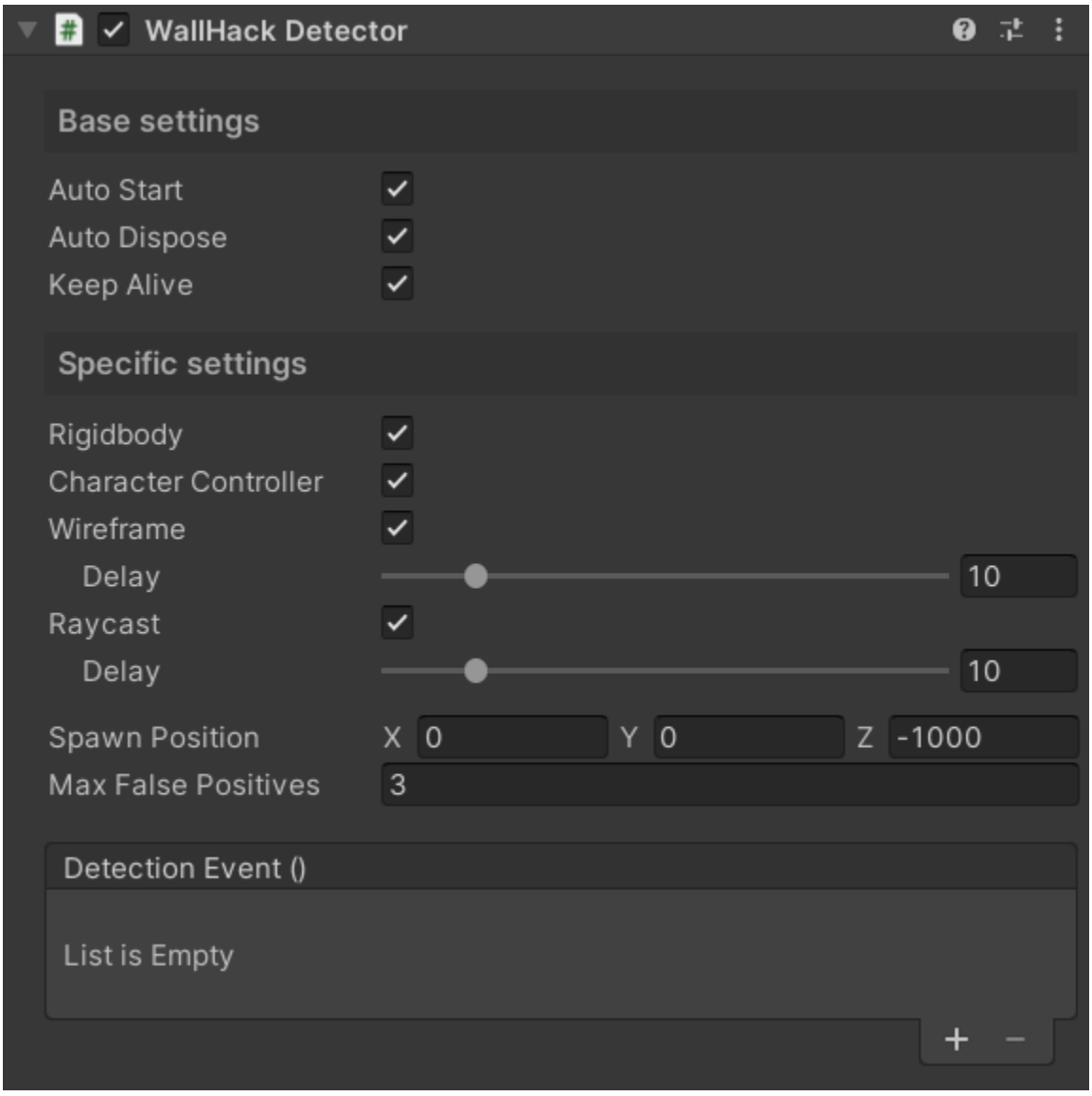
Configuration Guidelines

- Use **Head** request method for better performance, fall back to **Get** if server blocks head requests
- Set appropriate **thresholds** based on your game's requirements (e.g., 5 minutes for wrong time, 30 minutes for real cheat)
- Enable **Ignore Set Correct Time** to reduce false positives when users correct their system time
- Provide **fallbacks** for offline scenarios or network issues
- Consider using **manual checks** for critical moments (e.g., before important transactions)

> **WARNING**
>
> The detector requires Internet connection and appropriate 'android.permission.INTERNET' permission on Android. It automatically switches to the current domain on WebGL to avoid CORS limitations.

For detailed examples and API reference, see TimeCheatingDetector API documentation.

WallHack Detector



Combats three classes of wall hacks—walking through geometry, seeing through surfaces, and firing through walls.

Basic Usage

```
WallHackDetector.StartDetection(OnWallHackDetected);
```

Settings

Detection Modules

- **Rigidbody** – Check for "walk through the walls" cheats made via Rigidbody hacks. Disable to save resources if not using Rigidbody for characters.
- **Character Controller** – Check for "walk through the walls" cheats made via Character Controller hacks. Disable to save resources if not using Character Controllers.
- **Wireframe** – Check for "see through the walls" cheats made via shader or driver hacks (wireframe, color alpha, etc.). Uses specific shader under the hood. Disable to save resources if you don't care about such cheats.
- **Raycast** – Check for "shoot through the walls" cheats made via Raycast hacks. Disable to save resources if you don't care about such cheats.

Timing Settings

- **Wireframe Delay** – Delay between Wireframe module checks (1-60 seconds).
- **Raycast Delay** – Delay between Raycast module checks (1-60 seconds).

Detection Settings

- **Spawn Position** – World coordinates of the service container. Should be unreachable for your game objects to avoid collisions and false positives. Will have different active objects within 3x3x3 cube during gameplay.
- **Max False Positives** – Maximum false positives in a row for each detection module before registering a wall hack.

Advanced Features

Module Configuration

Configure specific detection modules based on your game's needs using the detector's properties (`CheckRigidbody`, `CheckController`, `CheckWireframe`, `CheckRaycast`).

Spawn Position Setup

Configure the service container position using `SpawnPosition` property to avoid conflicts with your game objects.

Custom Detection Parameters

Start detection with custom parameters using `StartDetection()` overloads that accept spawn position and max false positives.

Configuration Guidelines

- **Place spawn position** in an unreachable area (e.g., far coordinates like 1000, 1000, 1000)
- **Enable only needed modules** to optimize performance (disable unused detection types)
- **Adjust delays** based on your game's requirements (shorter delays = more detection, higher performance cost)
- **Set appropriate max false positives** to balance sensitivity and false positive reduction
- **Test thoroughly** as the detector creates service objects in your scene

> ⊗ **CAUTION**
>
> The detector creates service objects within a 3x3x3 cube at the spawn position. Make sure this area is completely empty and unreachable by your game objects to avoid false positives.

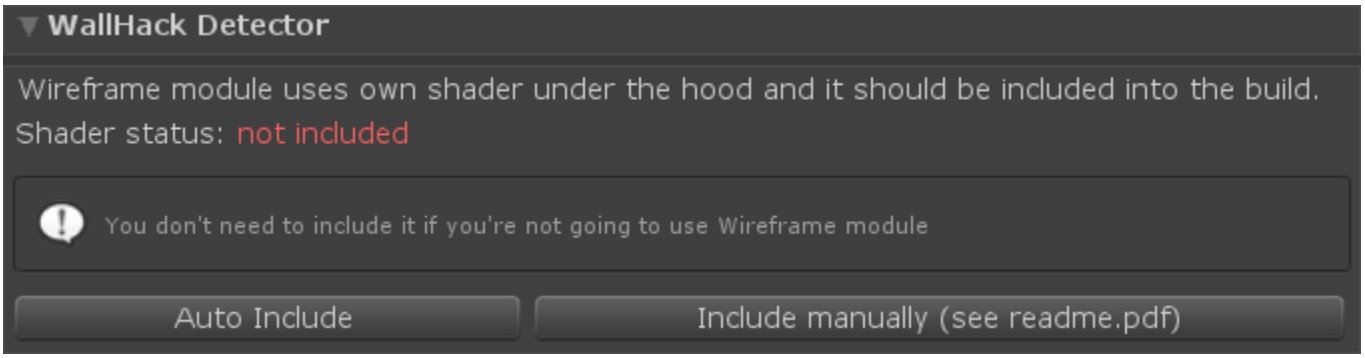
Debug Symbol

- `ACTK_WALLHACK_DEBUG` – Enables detailed debug logging to keep service renderers visible and provides additional detection details

Wireframe module shader setup

The Wireframe module uses `Hidden/ACTk/WallHackTexture` shader under the hood. This shader must be included in your build to exist at runtime. You'll see an error in logs at runtime if the shader is missing, and you'll be prompted in the Editor to include it when you run WallHackDetector without the shader.

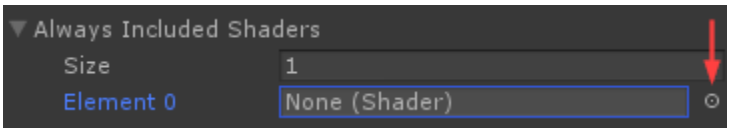
You can easily add or remove the shader via the **ACTk Settings** window. Press the **"Auto include"** button to automatically add the `Hidden/ACTk/WallHackTexture` shader to the Always Included Shaders list.



Alternatively, you may press the **"Include manually"** button to open Graphics Settings for your project and add the shader to the Always Included Shaders list manually.

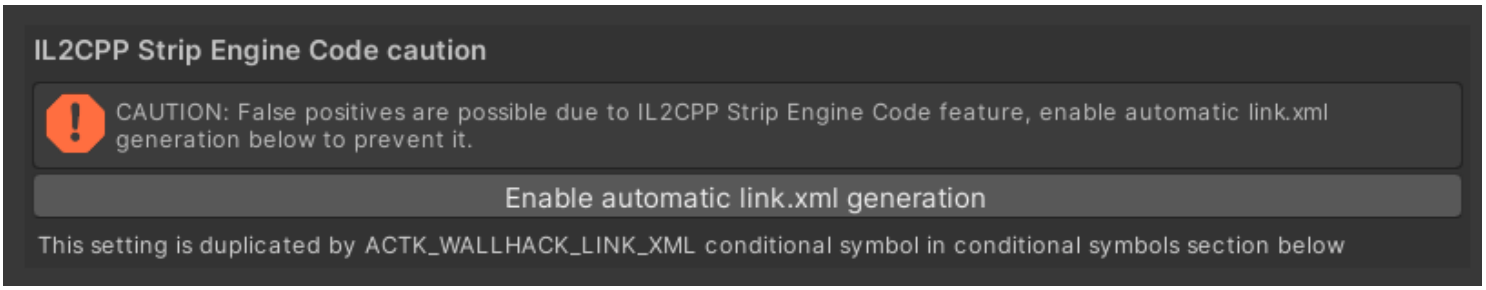
To manually add the shader to the Always Included Shaders list at the Graphics Settings:

1. Expand **Always Included Shaders**
2. Add one more element to the list
3. Click on the bulb next to the new element
4. Search for "wallhack" in the opened window and select the `WallHackTexture` with double-click



That's it for the wireframe module setup.

WallHack Detector and IL2CPP builds

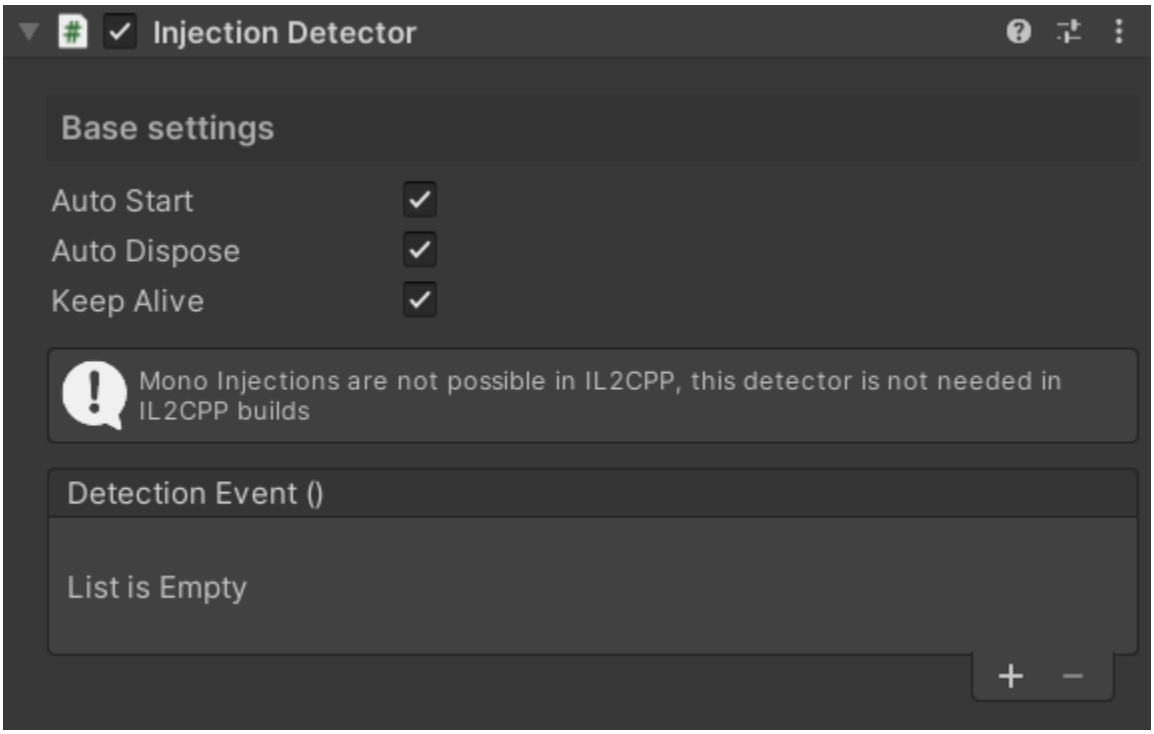


IL2CPP builds enable aggressive code stripping that can remove helper components the WallHack Detector relies on. Open `Tools > Code Stage > Anti-Cheat Toolkit > Settings`, locate the **WallHack Detector** section, and press **Enable automatic link.xml generation** so the required components are kept in your IL2CPP builds automatically.

> **TIP**
>
> - Wall hack cheats mostly target desktop first-person games. Disable unused modules to keep resource usage in check on other platforms.
> - The detector registers its service objects on the `Ignore Raycast` layer and keeps renderers disabled so they stay invisible during gameplay.
> - Spawn the service cube in an empty space away from level geometry and physics objects because the detector instantiates and moves helpers while it runs.

For detailed examples and API reference, see WallHackDetector API documentation.

Injection Detector



Protects Mono builds by monitoring loaded assemblies and comparing them to an expected whitelist.

Basic Usage

```
InjectionDetector.StartDetection(OnInjectionDetected);
```

Settings

- **Injection Detection Support** – Must be enabled in ACTk Settings to add mono injection detection support to builds. Has no effect for IL2CPP or unsupported platforms.
- **Custom Whitelist** – Fill any external assemblies which are not included into the project to the user-defined whitelist to make Injection Detector aware of them.

Platform Compatibility

- **Supported Platforms**: Standalone and Android builds only
- **IL2CPP**: Not available on IL2CPP-only platforms (Mono injections are not possible in IL2CPP)
- **WebGL**: Not supported due to platform limitations

> **NOTE**
>
> **Detection Scope**: This detector only detects **managed assembly injections** (Mono/.NET assemblies). **Native (unmanaged) code injections** are still possible and not covered by this detector. Native injections operate at a lower level (binary/DLL injections) and require additional protection mechanisms like native code obfuscation or anti-tampering solutions.

Advanced Features

Platform Support Check

Always check platform compatibility before starting detection using `InjectionDetector.IsSupported`.

Detailed Detection Callback

Use the detailed callback to get information about the injected assembly. The callback provides the name of the injected assembly for analysis.

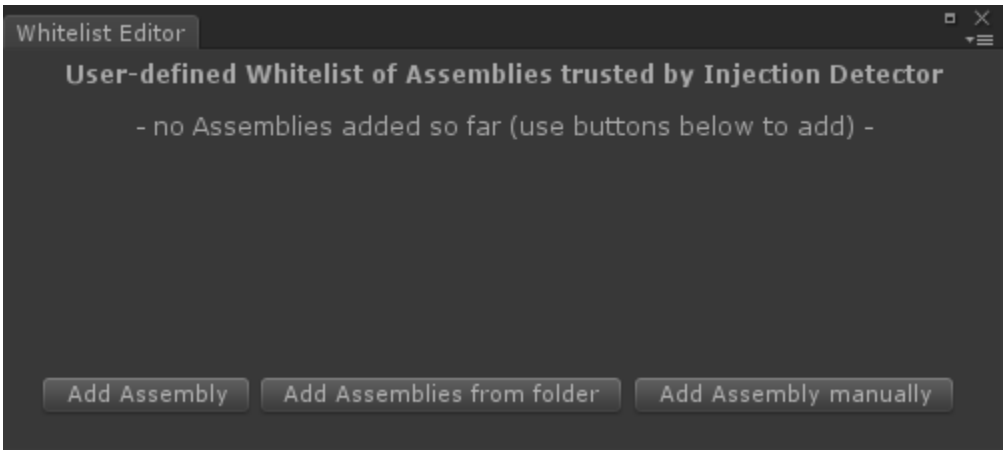
Whitelist management

The Injection Detector automatically adds assemblies that ship with your project to the whitelist during build. When you load assemblies from external sources or generate them at runtime, register them manually to prevent false positives.

Why the whitelist is needed:

- **External libraries**: Third-party DLLs loaded from outside the project
- **Runtime-generated assemblies**: Assemblies created dynamically at runtime
- **Plugin systems**: Assemblies loaded by plugin architectures
- **Modding support**: User-created assemblies loaded by mod systems

Use the whitelist editor to keep those assemblies on the safe list.



1. Open `Tools > Code Stage > Anti-Cheat Toolkit > Injection Detector Whitelist Editor` or press **Edit Whitelist** in the settings window.
2. Add assemblies with **Add Assembly**, **Add Assemblies from folder**, or **Add Assembly manually** depending on how you distribute plugins.
3. Remove entries with the `-` button or clear the list to rebuild it from scratch.

The whitelist is stored in `ProjectSettings/ACTkSettings.asset`, so share that file with your team to keep settings in sync.

> **NOTE**
> Enable `ACTK_INJECTION_DEBUG` when you need the detector to print the full name of a suspicious assembly and add it to the whitelist manually.

Configuration Guidelines

- **Enable injection detection** in ACTk Settings before building
- **Add external assemblies** to the whitelist to prevent false positives
- **Test on target platforms** before releasing (Standalone and Android only)
- **Check platform support** before starting detection to avoid runtime errors

> **TIP**
> This detector is disabled in the Editor by default due to specific assemblies causing false positives. Use `ACTK_INJECTION_DEBUG` symbol to force it in the Editor for testing.

Debug Symbols

The following conditional compilation symbols are available for debugging and development:

- **`ACTK_INJECTION_DEBUG`** – Enables basic debug logging and forces detection in the Editor for testing
- **`ACTK_INJECTION_DEBUG_VERBOSE`** – Provides additional detailed debug information for troubleshooting
- **`ACTK_INJECTION_DEBUG_PARANOID`** – Enables the most comprehensive debug logging for in-depth analysis

For detailed examples and API reference, see InjectionDetector API documentation.

Debug and Development

ACTk provides several conditional compilation symbols to aid in debugging and development. Enable these symbols in ACTk Settings (`Tools > Code Stage > Anti-Cheat Toolkit > Settings`).

> **NOTE**
>
> All debug symbols work only in development builds and the Editor.

Global Debug Symbols

ACTK_DETECTION_BACKLOGS

Enables detailed detection logging for all detectors. This symbol:

- Prints detection details to the console even when detectors are not running
- Helps identify false positives and troubleshoot detection issues
- Provides comprehensive logging including detection thresholds, counters, and timing information

Detector-Specific Debug Symbols

Individual detectors have their own debug symbols documented in their respective sections.

Handling detections

- Detectors stop themselves after firing when **Auto Dispose** is enabled. If you prefer continuous monitoring, disable Auto Dispose and restart the detector manually.
- Centralize cheat handling in a dedicated manager that can mute in-game input, display UI, and report analytics events.
- Combine detector output with `CodeHashGenerator` results and backend heuristics to minimize false positives before banning players.

Example implementations

- **Basic detector usage**: See `DetectorsExamples.cs` in `Examples/API Examples/Scripts/Runtime/UsageExamples/`
- **DOTS integration**: See `AntiCheatHost.cs` in `Examples/DOTS ECS Examples/Scripts/MonoBehaviors/`

Code integrity and platform helpers

Anti-cheat solutions work best when they are part of a broader hardening strategy. Use the tools below to make reverse engineering harder, validate that the shipped code matches your trusted build, and leverage platform APIs for additional safety nets.

Harden your builds

ACTk does not obfuscate or encrypt your managed assemblies on its own. Combine it with additional hardening layers to slow down reverse engineers and keep managed injections out of your builds.

- Prefer **IL2CPP** over Mono so your scripts are compiled into native binaries. IL2CPP removes the intermediate IL that dnSpy and similar tools rely on, blocks managed assembly injection, and forces attackers to use native disassemblers instead.
- Remember that IL2CPP still emits metadata for reflection. Attackers can recover class and method names with tools such as IL2CPP Dumper. Run a Unity-aware obfuscator before the IL2CPP step so symbol names become meaningless in the generated metadata.
- Evaluate native protectors when your threat model includes professional cheat makers. Solutions like Denuvo or VMProtect add runtime checks, while Unity-focused tools such as Mfuscator can encrypt IL2CPP metadata and add additional anti-tamper layers.
- Use `CodeHashGenerator` to confirm the shipped build matches the hashes produced in CI. Pair the client-side check with server validation for sensitive game flows.

Code hash pipeline

ACTk validates build integrity by generating hashes both in the editor and at runtime. This system helps ensure your shipped code matches your trusted build and hasn't been tampered with.

> **NOTE**
>
> Only Android and Windows PC builds are supported so far.

How hash generation works

Hash generation produces two types of hashes:

- **Per-file hashes**: Individual hashes for each file in the build
- **Summary hash**: A single hash derived from all per-file hashes

Both editor (via `CodeHashGeneratorPostprocessor`) and runtime (via `CodeHashGenerator`) operations produce these same hash types, though summary hashes may differ in some cases (e.g., Android App Bundle splits with platform-specific files).

Generate trusted hashes during builds

`CodeHashGeneratorPostprocessor` hooks into the build pipeline and produces reference hashes for each generated file. Enable **Generate code hash on build completion** in the settings window to automatically run it after every build.

You can also manually calculate external build hashes using:

- **Menu item**: `Tools > Code Stage > Anti-Cheat Toolkit > Calculate external build hashes`
- **Code**: `CodeHashGeneratorPostprocessor.CalculateExternalBuildHashes(buildPath, printToConsole)`

Validate builds at runtime

`CodeHashGenerator` computes hashes on the player device using two approaches:

Async approach (recommended):

```
// Note: This must be called from an async method
var result = await CodeHashGenerator.GenerateAsync();
if (result.Success)
    Debug.Log($"Summary Hash: {result.SummaryHash}");
```

Event-based approach:

```
CodeHashGenerator.HashGenerated += OnHashGenerated;
CodeHashGenerator.Generate();

private void OnHashGenerated(HashGeneratorResult result)
{
    if (result.Success)
        Debug.Log($"Summary Hash: {result.SummaryHash}");
}
```

Recommended validation strategy

1. **Compare summary hashes first** - if they match, the build is likely intact
2. **If summary hashes differ**, check per-file hashes against your pre-generated whitelist
3. **Treat unknown hashes as build alteration triggers** while ignoring absent files
4. **Implement server-side validation** for maximum security - send hashes to your server for comparison against trusted whitelist

CI/CD integration

For CI environments (like GitHub Actions) that don't properly wait for build completion:

```
// Use synchronous generation instead of relying on HashesGenerated event
CodeHashGeneratorPostprocessor.CalculateExternalBuildHashes(buildPath, printToConsole);
```

Platform considerations

- **Android App Bundles**: Summary hashes may differ per device split due to platform-specific files
- **Per-file hashes**: Should remain consistent between editor and runtime builds
- **Server validation**: Recommended for production use to prevent client-side tampering

For detailed API reference and examples, see CodeHashGenerator API documentation.

Example implementations

- **Basic usage**: See `GenuineChecksExamples.cs` in `Examples/API Examples/Scripts/Runtime/UsageExamples/`
- **Advanced validation**: See `GenuineValidatorExample.cs` in `Examples/Code Genuine Validation/Scripts/Runtime/`
- **DOTS integration**: See `UIActionSystem.cs` in `Examples/DOTS ECS Examples/Scripts/UI/Systems/`

Android installation source validation

`AppInstallationSourceValidator` helps you track the store that delivered your APK. Use it to detect installations from unofficial sources or device backups.

Basic Usage

```
var source = AppInstallationSourceValidator.GetAppInstallationSource();
if (source.DetectedSource == AndroidAppSource.Sideloaded)
    Debug.LogWarning("Sideloaded app detected!");
```

Advanced Features

- **Trust validation**: Check against trusted sources (Google Play, Galaxy Store, Amazon)
- **Security response**: Enable restricted mode for untrusted sources
- **Analytics integration**: Report installation sources to your backend
- **Error handling**: Detect access errors that may indicate security issues

For detailed API reference, see AppInstallationSourceValidator API documentation.

Example implementation

See `AndroidExamples.cs` in `Examples/API Examples/Scripts/Runtime/UsageExamples/` for complete usage examples.

Android screen recording blocker

`AndroidScreenRecordingBlocker` toggles the system-wide flag that prevents screenshots and screen recording on most stock ROMs. Use it to frustrate basic bots and discourage tool-assisted speed runs on non-rooted devices.

Basic Usage

```
AndroidScreenRecordingBlocker.PreventScreenRecording();
AndroidScreenRecordingBlocker.AllowScreenRecording();
```

Advanced Features

- **State-based control**: Block recording during sensitive gameplay, allow in menus
- **Platform detection**: Use `#if UNITY_ANDROID && !UNITY_EDITOR` for Android-only code
- **Lifecycle management**: Always allow recording when app is paused or destroyed

For detailed API reference, see AndroidScreenRecordingBlocker API documentation.

Example implementation

See `AndroidExamples.cs` in `Examples/API Examples/Scripts/Runtime/UsageExamples/` for complete usage examples.

> **TIP**
>
> Blocking is best-effort. Custom ROMs and rooted devices can bypass the flag, and the app preview disappears from the Android task switcher while prevention is active.

Integrations and ecosystem

ACTk plays well with popular Unity tooling. Use the integrations below to extend anti-cheat coverage without rewriting your own editor and runtime utilities.

Conditional symbols

Some add-ons expect the `ACTK_IS_HERE` compilation symbol. Enable it in the **Conditional Compilation Symbols** section of the settings window when you integrate third-party packages that look for ACTk.

Visual scripting

PlayMaker

ACTk ships a `Integration/PlayMaker.unitypackage` archive with custom actions:

- Access `ObscuredPrefs` and `ObscuredFilePrefs` without writing code.
- Control detectors (except the Obscured Cheating Detector) directly from state machines.
- Example flows live under `Scripts/PlayMaker/Examples`.

> **NOTE**
> PlayMaker does not allow custom variable types, so obscured primitives require manual integration. Refer to the community guide or download the integration example if you need them.

Get PlayMaker from the Asset Store.

Behavior Designer

Import `Integration/BehaviorDesigner.unitypackage` to unlock:

- Shared variables and tasks for Obscured types.
- Detector actions and conditions that slot into behaviour trees.
- Examples under `Scripts/BehaviorDesigner/Examples` demonstrate common patterns.

Get Behavior Designer from the Asset Store.

DOTS & hybrid projects

- Sample scenes live under `CodeStage/AntiCheatToolkit/Examples/DOTS ECS Examples` with a dedicated UI module (`Scripts/UI`).
- Systems such as `PlayerBootstrapSystem`, `CheatResponseSystem`, and `AntiCheatHost` show how to bridge ACTk with Entities.

- Authoring components expose Obscured data in the inspector while DOTS systems manipulate encrypted values safely.

Combine the DOTS sample with the Obscured types chapter for a production-ready starting point.

Additional plugins

These community solutions complement ACTk in production:

- **Mfuscator** – IL2CPP metadata encryption and native protection.
- **Simple IAP System / IAP Receipt Validator** – secure in-app purchase workflows by FLOBUK.
- **Cross-Platform Native Plugins (Essential Kit)** – cloud saves and platform integrations by Voxel Busters.
- **MFPS Anti-Cheat and Reporting** – multiplayer-friendly infrastructure by Lovatto Studio.
- **Android Native Pro** – platform hooks by Stan's Assets.

Share your integration to have it featured in future updates.

Troubleshooting guide

Use this section to resolve common issues quickly. When reporting bugs, include logs from a development build with `ACTK_DETECTION_BACKLOGS` enabled so the maintainer can inspect detector history.

Updates and migrations

- **Compilation errors after updating:** remove the previous `CodeStage/AntiCheatToolkit` folder before importing a new package to avoid stale assets.
- **Serialized obscured values behave incorrectly:** run the validators under `Tools > Code Stage > Anti-Cheat Toolkit > Validate` or use the migration commands under `Tools > Code Stage > Anti-Cheat Toolkit > Migrate`. Some releases ship model upgrades that need a one-time migration.

Detector-specific issues

- **False positives:** start detectors in `Start()` instead of `Awake()` or `OnEnable()`. If you're starting your existing in-scene detectors through code, make sure you're using `StartDetection()` methods at the MonoBehaviour's `Start()` phase, not at the `Awake()` or `OnEnable()` phases. All detectors except InjectionDetector will print details about detection if you enable `ACTK_DETECTION_BACKLOGS` flag in ACTk Settings. Inspect logs generated with `ACTK_DETECTION_BACKLOGS` to identify the module that triggered the detection. Please report any false positives with logs generated on development build with this flag enabled if possible.
- **Injection Detector false positives:** make sure you've added all external libraries your game uses to the whitelist (menu: `Tools > Code Stage > Anti-Cheat Toolkit > Injection Detector Whitelist Editor`). Enable `ACTK_INJECTION_DEBUG` to log offending assembly names.
- **WallHack Detector false positives:** make sure you've properly configured Spawn Position of the detector and the detector's service objects are not intersecting with any objects from your game. Also, make sure the `Ignore Raycast` layer collides with itself in the Layer Collision Matrix at `Edit > Project Settings > Physics`. WallHackDetector places all its service objects to the Ignore Raycast layer to help you avoid unnecessary collisions with your game objects and collides such objects with each other. That's why you need to make sure the Ignore Raycast layer will collide with itself. Add required components to a `link.xml` or let the settings window generate it for IL2CPP builds.
- **Obscured Cheating Detector false positives:** please ensure you have no corrupted instances of the serialized obscured types. To do so, navigate to `Tools > Code Stage > Anti-Cheat Toolkit > Validate` menu to detect corrupted data. Alternatively, reset broken variables in inspector and re-set them manually if you don't wish to migrate.
- **Time Cheating Detector errors on old Android devices:** if you see "Error response code: 0" or "java.io.EOFException" errors in logs from Time Cheating Detector on old Android devices (before Android 6), switch the request method from `HEAD` to `GET` to avoid UnityWebRequest bugs in pre-Android 6 runtimes.

Platform quirks

- **Android ProGuard / R8 errors:** add the following rules to `proguard-user.txt` (the settings window can generate the file via `Tools > Code Stage > Anti-Cheat Toolkit > Configure proguard-user.txt`):

```
-keep class net.codestage.actk.androidnative.ACTkAndroidRoutines { *; }
-keep class net.codestage.actk.androidnative.CodeHashGenerator { public void GetCodeHash(...); }
-keep class net.codestage.actk.androidnative.CodeHashCallback { *; }
```

- **ClassNotFoundException errors in ADB logs:** if you see "java.lang.ClassNotFoundException: net.codestage.*" errors in ADB logs, make sure to exclude native ACTk Android plugin from obfuscation or minification. See ProGuard rules above.
- **Android screen recording prevention not working:** the blocker relies on system APIs and may fail on heavily customized ROMs or rooted devices.
- **StreamingAssets access on Android/WebGL:** copy data from `StreamingAssets` into a writable location (such as `Application.persistentDataPath`) before calling `ObscuredFile` or `ObscuredFilePrefs`. This is an expected behavior due to the StreamingAssets nature at these platforms (see Unity Manual for more details).
- **IL2CPP component stripping:** when getting `Can't add component because class * doesn't exist` error, ensure required physics and rendering components exist in scenes or keep them via `link.xml` (WallHackDetector does use these Unity components: `BoxCollider`, `MeshCollider`, `CapsuleCollider`, `Camera`, `Rigidbody`, `CharacterController`, `MeshRenderer`). The settings window can auto-generate the file for WallHack Detector.
- **CodeHashGeneratorPostprocessor event not firing in CI:** call `CodeHashGeneratorPostprocessor.CalculateExternalBuildHashes(buildPath, printToConsole)` to run the process synchronously instead of relying on the asynchronous event.

Permissions and compliance

- **Unwanted Android INTERNET permission:** define `ACTK_PREVENT_INTERNET_PERMISSION` when you do not ship the Time Cheating Detector to avoid adding `android.permission.INTERNET`.
- **Unwanted Android READ_PHONE_STATE permission:** define `ACTK_PREVENT_READ_PHONE_STATE` when you do not use Device Lock based on platform identifiers. If you still enable Device Lock features with this symbol present, ACTk warns via logs and behavior is degraded.
- **Apple export compliance:** ACTk uses mass-market encryption by default. If you must avoid encryption declarations entirely, define `ACTK_US_EXPORT_COMPATIBLE` to downgrade algorithms (with reduced security, see Compatibility).

When in doubt, reach out via Discord or contact us with reproduction steps and logs.

Migration notes

Follow these guidelines when upgrading existing projects to new ACTk releases.

General tips

- Back up your project or commit to source control before importing a new package.
- Remove the previous `CodeStage/AntiCheatToolkit` folder so obsolete assets do not linger.
- After importing new version, run the validation and migration utilities under `Tools > Code Stage > Anti-Cheat Toolkit` to fix legacy serialized data, if you see any data corruption.

Upgrading from ACTk v2021

The v2022+ series redesigned several APIs:

- `CodeHashGeneratorPostprocessor.Instance` was removed. Access properties directly:
  - `Instance.callbackOrder` → `CodeHashGeneratorPostprocessor.CallbackOrder`
  - `Instance.HashesGenerated` → `CodeHashGeneratorPostprocessor.HashesGenerated`
- `HashGenerated` handlers now receive `IReadOnlyList<BuildHashes>` instead of arrays. Update your event signatures accordingly.
- `BuildHashes.FileHashes` and `HashGeneratorResult.FileHashes` changed from `FileHash[]` to `IReadOnlyList<FileHash>`.

Migrating serialized obscured data

Some releases updated the encrypted data model to close vulnerabilities. Use the utilities provided in the package when you upgrade from older versions:

- Menu commands under `Tools > Code Stage > Anti-Cheat Toolkit > Migrate` and `> Validate`
- Context menu commands in the Project window (`Code Stage > Anti-Cheat Toolkit > …`)
- Static helpers such as `MigrateEncrypted()` or `DecryptFromV0()` on specific obscured types

If you are coming from version 1.5.1.0 or earlier, request the 1.5.2.0 package to complete the intermediate migration step before jumping to modern releases.

When migrations are not viable

Reset serialized values in the inspector and reconfigure them manually. This is often faster when only a handful of fields are affected.

Platform compatibility

ACTk supports a wide range of Unity targets. Use this chapter to plan your deployment matrix and handle compliance requirements.

Feature availability

> **NOTE**
>
> This table lists features with platform-specific availability or requirements. Other ACTk features are generally available across all supported Unity targets unless a limitation is explicitly noted in their chapters.

Feature	Supported targets
Injection Detector	Android Mono, Standalone Mono
CodeHashGenerator	Windows Standalone, Android
AppInstallationSourceValidator	Android
AndroidScreenRecordingBlocker	Android

> **TIP**
>
> Prefer IL2CPP for platforms that support it. Even when a detector is limited to Mono, IL2CPP builds dramatically reduce the attack surface by removing the managed runtime.

Tested platforms

- Windows, macOS, Linux (Standalone builds)
- iOS
- Android
- UWP
- WebGL
- Consoles

Community reports also confirm compatibility with Windows Phone 8 and Apple TV.

Apple export compliance

ACTk uses "mass market" encryption (ObscuredFile, ObscuredFilePrefs, ObscuredLong/ULong/Double/Decimal) which is exempt from U.S. export licensing requirements under EAR 742.15.

Key Points:

- All ACTk features are compatible with Apple's export compliance and U.S. export regulations
- Does not require setting `ITSAppUsesNonExemptEncryption` plist.info key to YES
- These ACTk features use exempt ("mass market") encryption by default: ObscuredFile, ObscuredFilePrefs, ObscuredLong, ObscuredULong, ObscuredDouble, ObscuredDecimal
- You still need to declare encryption usage in App Store Connect, but no additional licensing is required

Alternative for stricter export requirements:

Define `ACTK_US_EXPORT_COMPATIBLE` if you must avoid the declaration entirely. This downgrades cryptography and causes partial ObscuredLong, ObscuredULong, ObscuredDouble and ObscuredDecimal obscuration weakening, making them not fully encrypted in some rare cases. Only enable it after evaluating the risk increase for your game.

Apple privacy manifest

ACTk does not call any native Required Reason APIs and does not ship its own privacy manifest. No additional action is required when you add the package to your project.

Android build tools

- Generate `proguard-user.txt` via `Tools > Code Stage > Anti-Cheat Toolkit > Configure proguard-user.txt` to preserve ACTk native classes when you enable minification.
- When you use `AndroidScreenRecordingBlocker`, verify the feature on real devices—some OEM builds ignore the prevention flag.

Third-party licenses

- **xxHashSharp** – BSD 2-Clause License (GitHub)
- **SharpZipLib** – MIT License (GitHub)

Support and resources

ACTk exists thanks to an active community of developers who report issues, suggest improvements, and share best practices. Stay in touch to keep your project protected.

Official channels

- Asset Store
- Homepage
- API reference
- Changelog
- Support

Community

- Discord
- Forum thread
- YouTube tutorials
- **Social media updates:**
  - Discord announcements
  - X (Twitter)
  - Facebook

Final thoughts

Anti-cheat is a marathon. Combine ACTk with strong server-side validation, frequent updates, and security reviews. Even large commercial anti-cheat providers cannot guarantee perfect protection—your goal is to block opportunistic cheaters and make targeted attacks expensive.

If ACTk saves you time, please consider leaving a review on the Asset Store. Feedback helps prioritize new features and keep the toolkit aligned with production needs.

— *Dmitry Yuhanov* (codestage.net)