

Hey there!

Huge thanks for purchasing **Anti-Cheat Toolkit (ACTk)**, comprehensive anti-cheat solution for Unity!

DISCLAIMER:

Anti-cheat techniques used in this plugin do not pretend to be 100% secure and unbreakable (this is impossible on client side), they should **stop most cheaters** trying to hack your app, though.
Please keep in mind: well-motivated and skilled hackers can break anything!



If you have questions about plugin API usage from the scripting side:

docs.codestage.net/actk/api/

Contents

- [Installation and setup](#)
- [Preventing memory cheats](#)
- [Preventing storage cheats](#)
- [Preventing Player Prefs cheats](#)
- [Player Prefs and Obscured Prefs Editor Window](#)
- [Common Device Lock feature notes](#)
- [Code obfuscation and overall code protection notes](#)
- [Android App Installation Source validation](#)
- [Android Screen Recording prevention](#)
- [Code Integrity validation](#)
- [Common cheats detectors features and setup](#)
- [Obscured types cheating detection](#)
- [Speed hack detection](#)
- [Time cheating detection](#)
- [Wallhacks detection](#)
- [Managed DLL Injection detection](#)
- [Third-party plugins integrations and notes](#)
- [Troubleshooting](#)
- [Migration notes](#)
- [Compatibility](#)
- [Final words from author](#)
- [Anti-Cheat Toolkit links and support](#)

Installation and setup

Before importing a new version into your project, please consider removing the previous one completely to avoid any issues due to possible file structure changes in the new version.

After importing plugin to the project, you will find new menus for further setup and control:

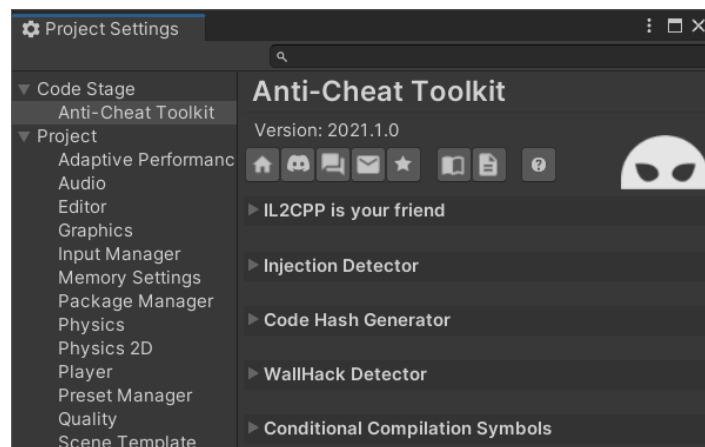
- **Tools > Code Stage > Anti-Cheat Toolkit**
- **GameObject > Create Other > Code Stage > Anti-Cheat Toolkit**

Please read through this document to know more about plugin features, best practices and hints.

Settings window

Use Editor Preferences section to configure detectors and conditional compilation symbols for debugging and compatibility purposes. All settings are stored at the "**ProjectSettings/ACTkSettings.asset**" of your project.

To open ACTk Settings, just use the **Tools > Code Stage > Anti-Cheat Toolkit > Settings** menu command.



Here you can see such items:

- ACTk version you're using in the project
- Useful links & shortcuts (homepage, Discord, forums, support, review, this manual, API reference, about)
- **IL2CPP is your friend** section – describes IL2CPP advantages over Mono from anti-cheat perspective & allows to easily switch to the IL2CPP if current platform supports it.
- **Injection Detector** settings section – see details at the [Managed DLL Injection detection](#) chapter.
- **Code Hash Generator** – see details at the [Code Integrity check](#) chapter.
- **WallHack Detector** settings section – see details at the [Wallhacks detection](#) chapter.
- **Conditional Compilation Symbols** section which allows switching different debug and compatibility tweaks, logs, and such.

Plugin features in-depth

Preventing memory cheats [\[video tutorial\]](#)

IMPORTANT: *Be careful while replacing regular types with the obscured ones – inspector values will reset!*

This is the most popular cheating method on different platforms. People use special tools to search variables in memory and change their values. It is usually money, health, score, etc. Just few examples: Cheat Engine, ArtMoney (PC), Game CIH, Game Guardian (Android). There are plenty of other tools for these and other platforms as well.

To leverage **memory** anti-cheat techniques just use **obscured** types instead of regular ones: [ObscuredInt](#), [ObscuredFloat](#), [ObscuredString](#) and so on (all basic types + few Unity-specific types are covered) ...

These types may be used instead of (and with) regular built-in types, just like this:

```
// place this line right at the beginning of yours .cs file!
using CodeStage.AntiCheat.ObscuredTypes;

int totalCoins = 500;
ObscuredInt collectedCoins = 100;
int coinsLeft = totalCoins - collectedCoins;

// will print: "Coins collected: 100, left: 400"
Debug.Log("Coins collected: " + collectedCoins + ", left: " + coinsLeft);
```

All `int` <-> [ObscuredInt](#) casts are done implicitly. Same for any other obscured type!

There is one big difference between regular `int` and [ObscuredInt](#) though – cheater will not be able to easily find and change values stored in memory as [ObscuredInt](#), what can't be said about regular `int`!

Well, actually, you may allow cheaters to find what they are looking for and catch them on cheating attempts. Actual obscured values are still safe in such case: the plugin will allow cheaters to find and change fake unencrypted variables if you wish them to ;)

For such cheating attempts detection use [ObscuredCheatingDetector](#) described below.

Obscured Types JSON Serialization

It's possible to serialize Obscured Types using multiple ways.

Newtonsoft Json.NET

If you are using [Newtonsoft Json.NET package](#) or pure Json.NET library, you can easily add Obscured Types **decrypted values** serialization using bundled [ObscuredTypesNewtonsoftConverter](#) class. You can use it as any other [JsonConverter](#):

```
using Newtonsoft.Json;
using CodeStage.AntiCheat.ObscuredTypes.Converters;

// add it to default converters one time globally
// -----
JsonConvert.DefaultSettings = ()=> new JsonSerializerSettings
{
    Converters = { new ObscuredTypesNewtonsoftConverter() }
};

var json = JsonConvert.SerializeObject(input);
// ...
var output = JsonConvert.DeserializeObject<T>(json);

// or add it to the reusable JsonSerializer Converters
// -----
var serializer = new JsonSerializer();
serializer.Converters.Add(new ObscuredTypesNewtonsoftConverter());
```

```

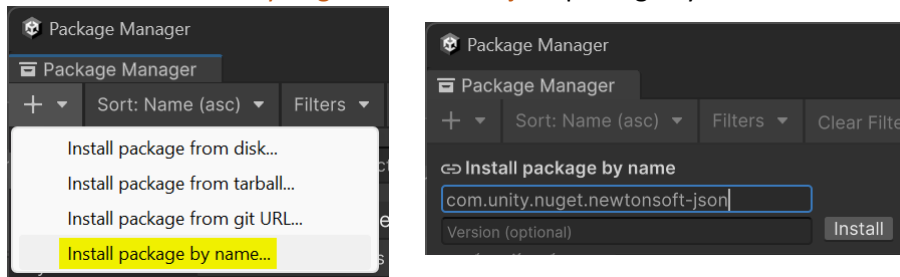
var json = serializer.Serialize(input);
// ...
var output = serializer.Deserialize<T>(json);

// or pass it directly every time you serialize / deserialize
// -----
var json = JsonConvert.SerializeObject(input, new ObscuredTypesNewtonsoftConverter());
// ...
var output = JsonConvert.DeserializeObject<T>(json, new ObscuredTypesNewtonsoftConverter());

```

IMPORTANT:

- [ObscuredTypesNewtonsoftConverter](#) is wrapped into the `#if ACTK_NEWTONSOFT_JSON` conditional compilation flag which is only enabled when your Unity project has [com.unity.nuget.newtonsoft-json package](#) of version 2.0.0 or newer added. If you're using pure Json.NET library, consider adding `ACTK_NEWTONSOFT_JSON` conditional symbol into the **Player Settings > Scripting Define Symbols** in order to manually enable this converter.
- You can add `com.unity.nuget.newtonsoft-json` package by name:



- As an alternative to the [JsonConverter](#), it's possible to implement [ISerializable](#) as shown in this [example](#).

JsonUtility

While it's less flexible, it's still possible to serialize Obscured Types with [JsonUtility](#) using the [SerializationCallbackReceiver](#) as shown in this [example](#).

Few more words about obscured types

You can find obscured types usage examples and even try to cheat them yourself in the scene "**Examples/API Examples**" shipped with the plugin.

Obscured types pro-tips:

- All simple Obscured types have public static methods **GetEncrypted()** and **SetEncrypted()** to let you work with encrypted value from an obscured instance. It may be helpful if you're going to use some custom saves engine.
- In some cases, cheaters can find obscured variables using so-called "unknown value" search. Despite the fact they'll find encrypted value and any cheating (freeze at in Cheat Engine or change in any memory editor) will be detected by [ObscuredCheatingDetector](#), they still can find it. To completely prevent this, you may use the special method in all basic obscured types – **RandomizeCryptoKey()**. Use it at the moments when variable doesn't change to change an encrypted representation keeping original value untouched. Cheater will search for unchanged value, but value actually will change, preventing variable finding.
- You may generate random crypto keys for the **Encrypt(value, key)** methods using **GenerateKey()** API. Do not forget to store those keys to **Decrypt()** your value later.

IMPORTANT:

- Currently, these types can be exposed to the inspector: [ObscuredBigInteger](#), [ObscuredBool](#), [ObscureDateTime](#), [ObscuredDecimal](#), [ObscuredDouble](#), [ObscuredFloat](#), [ObscuredInt](#), [ObscuredLong](#), [ObscuredQuaternion](#), [ObscuredShort](#), [ObscuredString](#), [ObscuredUInt](#), [ObscuredULong](#), [ObscuredVector2](#), [ObscuredVector2Int](#), [ObscuredVector3](#), [ObscuredVector3Int](#). **Be careful while replacing regular types with the obscured ones – inspector values will reset!**
- Obscured types are not compatible with Unity DOTS so please keep them outside your ECS entities ([example](#)).

- While still pretty fast and lightweight, obscured types are usually requiring additional resources comparing to the regular ones. Please try keeping obscured variables away from Updates, loops, massive arrays, etc. and keep an eye on your Profiler, especially when working on mobile projects.
- [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.
- Generally, I would suggest casting obscured variables to the regular ones to make any advanced operations and cast it back after that to make sure it will work fine.

Preventing storage cheats [[video tutorial](#)]

Obscured File

Files are common place for saved games and other sensitive information storage. But without proper protection, files are vulnerable to these threats:

- Sensitive data identification at non-binary files (game variables, credentials, in-game dev cheats etc.).
- Data modification (to cheat money from 100 to 999999 etc.).
- Cheated / modified file sharing with other cheaters.

[ObscuredFile](#) does cover all these threats:

- Optionally encrypts data to hide it from curious eyes.
- Has built-in tamper detection, so data modification can be spotted (both for encrypted and plain files).
- Optionally locks data to the Device ID or user-defined ID to detect files shared from other devices. See [device lock notes](#) to know more.

ObscuredFile can read and write raw byte arrays only. Check out [ObscuredFilePrefs](#) below for user-friendly PlayerPrefs – alike API.

Setup and usage are very straightforward:

```
// place this line right at the beginning of yours .cs file
using CodeStage.AntiCheat.Storage;

// create new instance with default settings
var secureFile = new ObscuredFile();

// write data
var writeResult = secureFile.WriteAllBytes(abstractBytes);
if (writeResult.Success)
    Debug.Log($"Data saved to {secureFile.FilePath}");
else
    Debug.LogError($"Couldn't save data: {writeResult.Error}");

// read data
var readResult = secureFile.ReadAllBytes();
if (readResult.Success)
    // process readResult.Data;
else if (readResult.CheatingDetected)
    // punish cheaters, check DataFromAnotherDevice \ DataIsNotGenuine readResult properties
else
    Debug.LogError($"Something went wrong while reading data: {readResult.Error}");
```

In the example above, abstractBytes are saved and loaded with default settings, but you are free to configure file path and name, encryption settings, lock to device feature settings and anti-tamper using [ObscuredFileSettings](#) APIs.

To know more about how to configure and use [ObscuredFile](#), please refer to the [API documentation](#) and usage example: [Examples\API Examples\Scripts\Runtime\UsageExamples\ObscuredFilePrefsExamples.cs](#)

IMPORTANT:

- Call [UnityApiResultsHolder.InitForAsyncUsage\(true\)](#); from main thread before working with obscured files from background threads.
- Please keep in mind, [ObscuredFile](#) will work slower comparing to the regular [File](#) due to additional encryption. Make sure to avoid using it at hot paths.

Obscured File Prefs

This is a static [ObscuredFile](#) wrapper with easy to use generic API. It allows reading and writing prefs of different types (all basic C# types, [BigInteger](#), [byte\[\]](#), [DateTime](#) and few Unity types are supported).

It has all the same features [ObscuredFile](#) has and does store prefs in an [ObscuredFile](#), protected from cheaters. Here is a simple usage example:

```
// place this line right at the beginning of yours .cs file
using CodeStage.AntiCheat.Storage;

// init with default settings
ObscuredFilePrefs.Init();

// listen for cheating
ObscuredFilePrefs.NotGenuineDataDetected += OnDataCheat;
ObscuredFilePrefs.DataFromAnotherDeviceDetected += OnLockCheat;

// write pref
ObscuredFilePrefs.Set("pref key", data);

// read pref
data = ObscuredFilePrefs.Get("pref key", defaultValue);

// alternatively, specify data type and omit defaultValue argument
// data = ObscuredFilePrefs.Get<T>("pref key");

// get all pref keys to iterate when needed
var keys = ObscuredFilePrefs.GetKeys();
```

Similarly to [ObscuredFile](#) example, default settings were used, and you can configure [ObscuredFilePrefs](#) for your needs with [ObscuredFileSettings](#) APIs.

IMPORTANT:

- Call [UnityApiResultsHolder.InitForAsyncUsage\(true\)](#); from main thread before working with obscured files from background threads.
- Please keep in mind, [ObscuredFilePrefs](#) will work slower comparing to the regular [File](#) or [PlayerPrefs](#) due to additional encryption. Make sure to avoid using it at hot paths.
- [ObscuredFilePrefs](#) has [AutoSave](#) feature which is enabled by default. It prevents possible loss of unsaved data and saves it on app quit (desktops) and app loosing focus (mobiles). You can disable it at [ObscuredFileSettings](#) on your own risk if you wish to have full control.

Preventing Player Prefs cheats [[video tutorial](#)]

Unity developers often use [PlayerPrefs \(PP\)](#) class to store small amounts of sensitive data (money, game progress etc.), but it can be found and tampered with almost no effort. This is as simple as opening regedit and navigating it to the [HKEY_CURRENT_USER\Software\Your Company Name\Your Game Name](#) on Windows, for example!

That's why I decided to include [ObscuredPrefs \(OP\)](#) class into this toolkit. It allows you to save data as usual, but keeps it safe from views and changes, exactly what we need to keep our saves cheaters proof! 😊

Here is a simple example:

```
// place this line right at the beginning of yours .cs file
```

```
using CodeStage.AntiCheat.Storage;
```

```
ObscuredPrefs.Set<float>("currentLifeBarObscured", 88.4f);  
var currentLifeBar = ObscuredPrefs.Get<float>("currentLifeBarObscured");  
  
// will print: "Life bar: 88.4"  
Debug.Log("Life bar: " + currentLifeBar);
```

As you can see, nothing changed in how you use it – everything works just like with good old regular [PlayerPrefs](#) class, but now your saves are secure from cheaters (well, from most of them)!

Additional functionality:

- You may subscribe to the **NotGenuineDataDetected** event. It allows you to know about saved data integrity violation. Fires once (for whole session) as soon as you read some tampered data.
- [OP](#) supports plenty of additional data types comparing to regular [PP](#): all basic C# types, [BigInteger](#), [byte\[\]](#), [DateTime](#) and few Unity types are supported.
- Device Lock feature is supported. See [device lock notes](#) to know more.

ObscuredPrefs pro-tips:

- You may easily migrate from [PP](#) to [OP](#) – just replace [PP](#) occurrences in your project with [OP](#), and you're good to go (be careful, though, don't replace [PP](#) with [OP](#) in the ACTk classes)! [OP](#) automatically encrypts any data saved with [PP](#) on first read. Original [PP](#) key will be deleted by default. You may preserve it though using **preservePlayerPrefs** flag. In such case [PP](#) data will be encrypted with [OP](#) as usual, but the original key will be kept, allowing you to read it again using [PP](#). You may see **preservePlayerPrefs** in action in the [API Examples](#) scene.
- You may mix regular [PP](#) with [OP](#) (make sure to use different key names though!), use obscured version to save only sensitive data. No need to replace all [PP](#) calls in project while migrating, regular [PP](#) works faster comparing to [OP](#).
- Use [OP](#) to save adequate amount of data, like local leaderboards, player scores, money, etc., avoid using it for storing massive portions of data like maps arrays, your database, etc. – use [ObscuredFile or ObscuredFilePrefs](#) for that.
- There are some great contributions extending regular [PlayerPrefs](#) around, like [ArrayPrefs2](#) for example. [OP](#) could easily replace [PP](#) in such classes, making all saved data secure.

IMPORTANT:

- Please keep in mind [OP](#) will work slower comparing to the regular [PP](#) since it encrypts and decrypts all your data, consuming some additional resources. You are welcome to check it yourself, using [Performance Obscured Tests](#) component on the [PerformanceTests](#) game object in the [API Examples](#) scene.
- Please consider using [ObscuredPrefs.DeleteAll\(\)](#) to remove all prefs instead of [PlayerPrefs.DeleteAll\(\)](#) to properly clear internals and avoid any data loss when saving new obscured prefs after [DeleteAll\(\)](#) call.

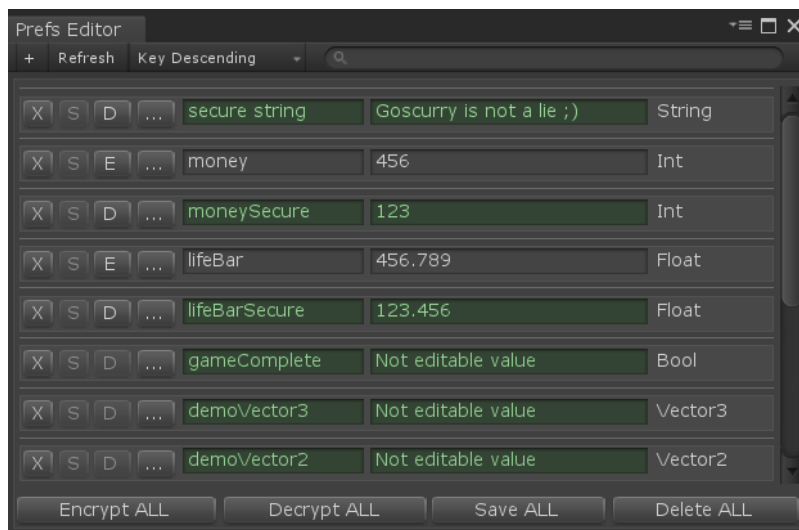
ObscuredPrefs / PlayerPrefs editor window

ACTk includes helpful Editor Window – **Prefs Editor**.

Use it to edit your PlayerPrefs and ObscuredPrefs in Unity Editor.

Open it via menu command:

[Tools](#) > [Code Stage](#) > [Anti-Cheat Toolkit](#) > [Prefs Editor as Tab](#) (utility window mode also available)



Quick tour:

- "+" button: add new pref – select type, optionally enable encryption, set key and value, press OK button
- "Refresh" button: re-read all prefs and update list
- "Key Descending" - sorting type, other possible values: Key Ascending, Type, Obscure
- search field – just start typing, and it will filter out records with name containing specified text
- list of both ObscuredPrefs and PlayerPrefs with paging (shows 50 records per page)
- record breakdown:
 - "X" button – deletes record from prefs
 - "S" button – saves changes to the prefs storage
 - "E" / "D" buttons – encrypt / decrypt pref (obscured prefs are colored in green)
 - "..." button – extra actions, like Copy pref to clipboard
 - key and value fields allow changing the key or value of the pref (if pref has editable type)
 - type label shows pref type
- some buttons in bottom for group actions (with self-explanatory names)
- has overwrite confirmation
- has prefs parse progress bar if you're working with huge prefs amount (over 1k)
- works on Win, Mac, Linux

Prefs Editor ignores few standard prefs Unity uses for own needs (like UnitySelectMonitor, UnityGraphicsQuality, etc.).

Please note, on Windows, prefs stored in Editor and in standalone player are placed in different locations, so it's not possible to use Prefs Editor for prefs saved outside the Unity Editor.

Common Device Lock feature notes

When using the **Device Lock** feature with [ObscuredPrefs](#), [ObscuredFile](#) or [ObscuredFilePrefs](#), please consider the following:

- **DeviceLockSettings** property allows locking any saved data to the current device. This could be helpful to prevent save games moving from one device to another (saves with 100% game progress, or with bought in-game goods, for example).

WARNING: On iOS, use at your peril! There is no reliable way to get a persistent device ID on iOS. So, avoid using it or use with **DeviceIdHolder.DeviceId** to set own device ID (e.g., user email).

You may specify three different levels of data lock strictness:

None: you may read both locked and unlocked data, saved data will remain unlocked.

Soft: you still may read both locked and unlocked data, but all saved data will lock to the current device.

Strict: you may read only locked to the current device data. All saved data will lock to the current device.

See [DeviceLockSettings](#) description in the [API docs](#) for more info.

Android users: see **Troubleshooting** section below if you don't use this feature.

- Listen to the **DataFromAnotherDeviceDetected** event to detect data from another device. In ObscuredPrefs invokes only once (per session), in ObscuredFile \ ObscuredFilePrefs – invokes every time a file is read.

- When [DeviceLockLevel.Strict](#) is used, saves from other devices are not readable by default. You may reduce [DeviceLockSettings.DeviceLockTamperingSensitivity](#) to [Low](#) (detection event still will be fired) or [Disabled](#) (event will not be fired). This can also be used to restore data in case Device ID changed unexpectedly.
- First access to the Device ID can produce a CPU spike on some platforms, causing gameplay hiccup when you first time load or save data. To avoid this, call **DeviceldHolder.ForceLockToDeviceInit()** to avoid possible spike on first data load / save with **Device Lock** enabled. Use this method to force getting device ID at the desired time, while you're showing something static, like a loading fade or splash screen.
- Use **DeviceldHolder.Deviceld** property to explicitly set device ID. This may be useful to lock saves to the unique user ID (or email) if you have server-side authorization. Best for iOS, since there is no way to get a consistent device ID (on iOS 7+), all we have is Vendor and Advertisement IDs, both can change and have restrictions \ corner cases.

Code obfuscation and overall code protection notes

ACTk does not obfuscate or protect your source code, and thus it's still vulnerable to the hackers and reverse-engineers even if you are using all ACTk features.

Thus, it's highly recommended to use **both** a good reverse-engineering protection (code obfuscator or/and native protector) and an IL2CPP scripting backend when possible, to make it much harder to analyze and change your compiled application code.

IL2CPP does produce raw binary code with metadata instead of Mono's IL byte code making all IL reversing tools useless (like [dnSpy](#) and such) and making it much, much harder to get good decompilation of your method bodies (only possible with expensive and difficult to use native disassemblers and decompiles). As a side bonus, it also totally prevents managed assemblies' injection into the Mono ApplicationDomain.

Metadata is generated for the limited reflection purposes and still allows constructing IL assemblies without method code but with all namespaces, classes, fields and such and cheaters do actively use tools like [IL2CPP Dumper](#) to reconstruct IL assemblies to look at the overall code structure.

That's why it's dramatically important to complement IL2CPP with a good code obfuscator which has the names obfuscation feature and is integrated into the Unity's build process to obfuscate code **before** IL2CPP makes a build. In such case, most of IL2CPP metadata will become a mess of useless names, making reconstructed IL assembly very hard to reverse-engineer and analyze.

There are lots of good code obfuscators around, including free ones, though not that much are properly integrated with Unity, so make sure obfuscator of your choice do support Unity IL2CPP builds obfuscation.

It's also important to ship the initial version obfuscated, since there are tools which can automatically deobfuscate new binaries versions based on older non-obfuscated version.

There are also plenty of native protectors around, for specific platforms and use cases (examples: Denuvo, VMProtect, etc.).

I'd like to highlight Unity-specific protector for IL2CPP builds which does encrypts IL2CPP metadata and adds few other layers of native protection to your app: [Mfusator](#) (read more at the [third-party](#) section).

Another solution to cover you from the reverse-engineering threat on Android platform I can recommend: [Skyforce](#).

Anyway, it's always a good idea to check if your code is genuine and was not tampered if you have some sensitive parts of the code on the client side (i.e., not on the server side which is not reachable to hackers), and ACTk is here to help with [CodeHashGenerator](#).

Android App Installation Source validation

ACTk has the [AppInstallationSourceValidator](#) tool to easily figure out your Android application installation source.

It's as simple to use as just calling [AppInstallationSourceValidator.IsInstalledFromGooglePlay\(\)](#) to confirm your app was installed from the Google Play Store.

If you need more control, you can gather installation source package name and corresponding guessed installation source enum value with calling the [AppInstallationSourceValidator.GetAppInstallationSource\(\)](#) API:

```
// place this line right at the beginning of yours .cs file!
```

```
using CodeStage.AntiCheat.Genuine.Android;

// getting and processing installation source
var source = AppInstallationSourceValidator.GetAppInstallationSource();

if (source.DetectedSource != AndroidAppSource.AccessError)
{
    Debug.Log($"Installed from: {source.DetectedSource} (package name: {source.PackageName})");
    if (source.DetectedSource == AndroidAppSource.SamsungGalaxyStore)
        Debug.Log("App was installed from the Galaxy Store!");
}
else
{
    Debug.LogError("Failed to detect the installation source!");
}
```

Android screen recording prevention

ACTk has the [AndroidScreenRecordingBlocker](#) tool to prevent screenshot and screen recording of your app which can be useful to resist some bots on non-rooted devices.

IMPORTANT:

- While Android makes its best to prevent screenshots and video recording, it's not guaranteed it will work with some custom ROMs built-in software.
- When prevention is on, app preview is not available in Task Manager too.

Here is a simple usage example:

```
// place this line right at the beginning of yours .cs file!
using CodeStage.AntiCheat.Uutils;

private void SwitchScreenRecordingPrevention(bool prevent)
{
    if (prevent)
        AndroidScreenRecordingBlocker.PreventScreenRecording();
    else
        AndroidScreenRecordingBlocker.AllowScreenRecording();
}
```

Code Integrity validation

IMPORTANT:

- *Only Android and Windows PC builds are supported so far*
- *For advanced Unity developers mostly as it requires additional coding*

The general idea behind integrity validation is to make a unique fingerprint (hash) of compiled binaries, which will change as soon as binaries will be altered after you get the fingerprint, and to compare that fingerprint against the current fingerprint your runtime application has.

When you have to deal with novice cheaters, it's enough to make a comparison right in your application in most simple cases and more likely it will not be noticed by the novice cheater, so you're free to start from simple hash comparison right in the C# code of your application.

However, due to the nature of the threat, your check can be altered too by the more advanced cheaters, so it's more secure to make a validation outside your C# code. The best choice here – make a server-side validation. You just send current hashes to the server and check them against initially generated (whitelisted) ones to make sure your code didn't change.

So, your validation procedure generally consists of three steps: get the *genuine* hashes, get the *actual runtime* hashes, compare hashes. Please read more details about each step below.

IMPORTANT:

- Hashes generation operation (both in editor or at runtime) produces *per-file hashes* for each file in build and gets a **summary hash** from those files.
- Thus, while per-file hashes should remain unchanged both in editor (got from [CodeHashGeneratorPostprocessor](#)) and runtime (got from [CodeHashGenerator](#)), summary hash may differ in some cases (for example, when your runtime app is a part of AAB bundle, and it has no few platform-specific files inside).
- Recommended hash comparison strategy: compare summary hashes first and if they don't match, check all runtime per-file hashes against pre-generated per-files hashes whitelist (and treat any unknown hash as a build alteration trigger while ignoring absent files).

CodeHashGeneratorPostprocessor

This Editor script allows generating *genuine* hashes of your code in your resulting build file(s).

You can listen to the [HashesGenerated](#) event to get the resulting hashes after each build.

Enable "[Generate code hash on build completion](#)" option in the [ACTk Settings](#) to enable this event. Resulting hashes will also be printed to the Editor Console (even if you didn't subscribe to the event).

You can also get any external build code hashes manually via built-in menu item: [Tools > Code Stage > Anti-Cheat Toolkit > Calculate external build hashes](#)

Or you can call the **CodeHashGeneratorPostprocessor.CalculateExternalBuildHashes()** directly from your Editor code for the same result.

This postprocessor was made to generate hashes without executing builds at runtime.

Use [CodeHashGenerator](#) in your genuine build to generate *genuine* hashes at runtime if you encounter any problems with in-editor hashes generation or in case you post-process your code after finishing Unity build.

See usage example in the Editor *PC Build* Hashing example counterpart placed at "[Examples/Code Genuine Validation/Scripts/Editor](#)".

CodeHashGenerator

This Runtime script allows generating hashes right from your running app.

Call the awaitable **GenerateAsync()** which do return hashes or subscribe to the static event **HashGenerated** and call **Generate()** method to start hash generation. Underlying hash generation code differs for different platforms, but in general, it does works at the separate threads or coroutines when possible, to make the generation process smooth and avoid CPU spikes, preventing overall app performance degradation.

You are welcome to generate your first *genuine* hashes at runtime from your trusted device to compare them later against runtime hashes from your user's devices to validate code integrity. But please note in case of Android App Bundle usage, hashes got from the app running on device will differ depending on your device hardware since AAB gets split into different APKs by Google Play. In such case the best strategy is to calculate hashes from the initial aab build to get all genuine files hashes.

See usage example in the Runtime *PC Build* Hashing example counterpart placed at "[Examples/Code Genuine Validation/Scripts/Runtime](#)" and "[Examples/Code Genuine Validation/GenuineValidator](#)" scene.

IMPORTANT:

The example provided is for the Windows PC platform only, since it's a better fit to demonstrate the concept and keep it simple.

Common detectors features and setup

ACTk has various *detectors* included to let you detect different cheats and react accordingly. All detectors have some common features and setup processes.

Generally, you have three ways to set up and use any detector:

- setup through the Unity Editor
- setup through the code
- mixed mode

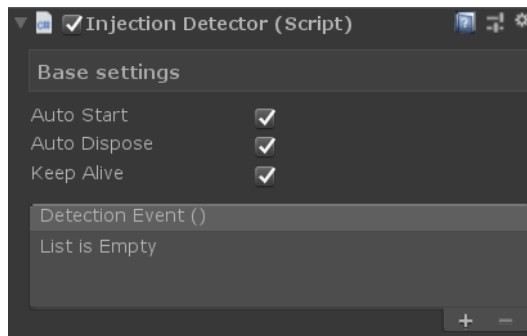
Setup through the Unity Editor

First, you need to add a detector to the scene. You have two options for that:

1. Use **GameObject > Create Other > Code Stage > Anti-Cheat Toolkit** menu to quickly add a detector to the scene. "Anti-Cheat Toolkit Detectors" Game Object with detector will be created in the scene (if it does not exist). This is the recommended way to set up detectors.
2. Alternatively, you may add a detector to any existing Game Object of your choice via the **Component > Code Stage > Anti-Cheat Toolkit** menu

Next step – configure added detector. All detectors have some common options to configure.

Here is the freshly added **Injection Detector** for example (as it has no other options except the common ones):



Auto Start: enable to let detector start automatically after scene load. Otherwise, you'll need to start it explicitly from code (see details below). To use this feature, you should configure the *Detection Event* described below.

Auto Dispose: enable to let detector automatically dispose itself and destroy own component after cheat detection. Otherwise, the detector will just stop.

Keep Alive: enable to let detector survive new level (scene) load. Otherwise, the detector will self-dispose on new level load.

Detection Event: it's a standard [Unity Event](#) which detector executes on cheat detection.

Recommended setup – keep all options enabled and set appropriate event for detection.

In such case, you'll have always working detector travelling through the scenes, and it will destroy itself after detection. You may attach your script with detection callbacks to the same Game Object where you have your detector, and they both will survive scene reload.

Setup through the code

If you prefer to set up things from code – just call the static **StartDetection(callback)** method of any detector once anywhere in your code to get that detector running with default parameters. The detector will be added to the scene automatically if it doesn't exist.

If you wish to tune any options – just use the static **Instance** property to reach all configurable fields and properties right after starting the detector (Instance will be null if there is no such detector in the scene).

Some specific for detector options usually available for initial configuration through the arguments of the overloaded **StartDetection()** methods, see the API docs for your detector to figure out which options are available for the initial tuning.

Here is an example for the **Injection Detector**:

```
// place this line right at the beginning of yours .cs file!
using CodeStage.AntiCheat.Detectors;

// starts detection with specified callback
InjectionDetector.StartDetection(OnInjectionDetected);

// this method will be called on injection detect
```

```
private void OnInjectionDetected() { Debug.Log("Gotcha!"); }
```

Mixed setup

Detectors allow you to set up them in a mixed way – combining configuration in the inspector and manual launch through the code.

You may add a detector to the scene as described in the **Setup through the Unity Editor** topic, keeping the Detection Event empty and start it manually from the code using **StartDetection(callback)** method as described in the **Setup through the code** topic.

This way allows you to control the moment when the detector should start and lets you configure the detector both from the Inspector and from the code through the **Instance** property before starting it.

To use such an approach, you should disable **Auto Start** option in the detector's inspector.

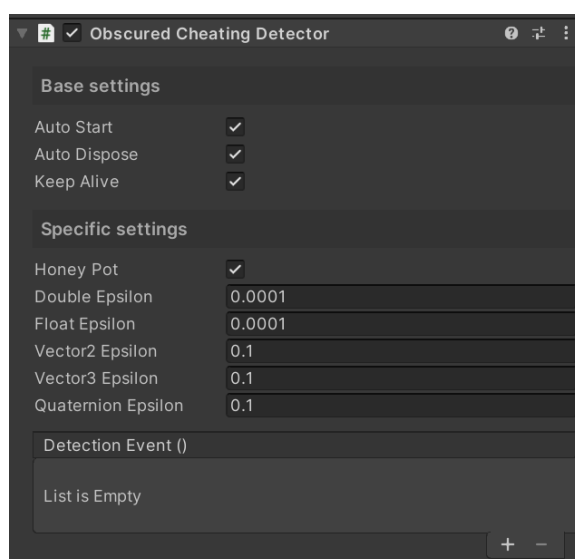
You also may fill Detection Event in the inspector and start detector using **StartDetection()** method (without arguments).

Obscured types cheating detection

Anti-Cheat Toolkit's **Obscured Cheating Detector** allows detecting cheating of **all** obscured types except **ObscuredPrefs** (it has own detection mechanisms covered in the next section).

When running, this detector allows obscured vars to use fake unencrypted values as a honeypot for cheaters. They'll be able to find desired values, but changing or freezing them will not do anything except triggering cheating detection.

See [Common detectors features and setup](#) topic above for the details on setup and common features of any detector.



Honey Pot: creates fake decrypted values for cheaters to find and hack it, triggering cheating attempts detection. Disable to make it harder to reveal obscured variables in memory, or keep enabled to catch more casual cheaters.

Epsilons: maximum allowed difference between fake and real encrypted variables before cheat attempt will be detected. Default values are suitable for most cases, but you're free to tune them for your case (if you have false positives for some reason, for example).

IMPORTANT:

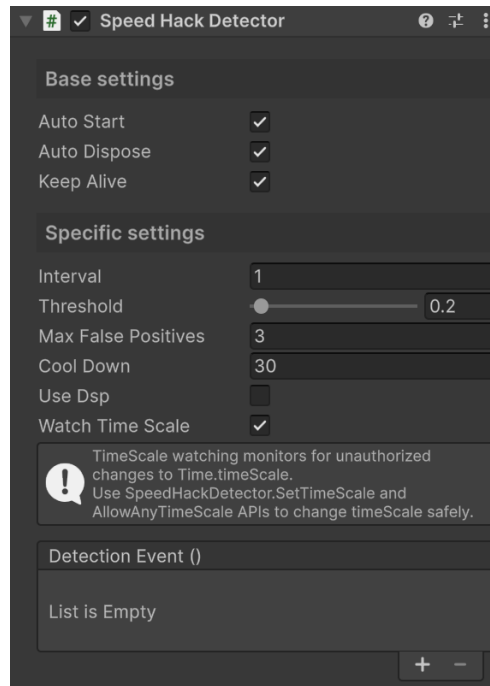
- You won't be able to react to the cheating detection while the detector is not running. ACTk will still print the detection details to the console even if the detector is not running or doesn't exist though if you'll use the **ACTK_DETECTION_BACKLOGS** setting, in the Development builds and in the Editor.
- It's possible to get an extra context on latest detection event using the **ObscuredCheatingDetector.Instance.LastDetectionInfo** property.

Speed hack detection [[video tutorial](#)]

This type of cheating is very popular and used pretty often since it is straightforward to use and any child around may try it on your game. Speed Hack allows speeding up or slowing down your game in various cheating tools with a few simple clicks.

Anti-Cheat Toolkit's **Speed Hack Detector** allows detecting speed hack usage (from tools like Cheat Engine, Game Guardian, etc.).

See **Common detectors features and setup** topic above for the details on setup and common features of any detector.



Interval: detection period (in seconds). Better, keep this value at one second and more to avoid extra overhead.

Threshold: allowed speed multiplier, both for speeding up and slowing down. It was introduced to decrease possible false positives for rare cases with slightly faster or slower timers due to the hardware diversity. The default value of 0.2 allows both 1.2x and 0.8x game speeds, which in general is stable enough and should work around most known timers' problems.

Max False Positives: in some very rare cases (system clock errors, OS glitches, etc.) detector may produce false positives, this value allows skipping a specified amount of speed hack detections before reacting on cheat. When actual speed hack is applied – the detector will detect it on every check. Quick example: you have Interval set to 1 and Max False Positives set to 5. In case speed hack was applied to the application, the detector will fire detection event in ~5 seconds after that (Interval * Max False Positives + time left until next check).

Cool Down: allows resetting internal false positives counter after a specified number of successful shots in the row. It may be useful if your app has rare yet periodic false detections (in case of some constant OS timer issues for example), slowly increasing false positives counter and leading to the false speed hack detection at all. Set the value to 0 to completely disable the cool down feature.

Use Dsp: activates DSP Audio module which allows catching speed hacks in sandboxed environment (like WebGL, VMs with built-in speed hacks and such). Off by default.

WARNING: due to extra sensitivity, it can lead to false positives on some hardware, especially when running at iOS. Always test against false positives on target devices, 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\(\)](#) or [AllowAnyTimeScale\(\)](#) APIs to safely change timeScale without triggering false positives.

WARNING: May cause false positives if you change timeScale directly.

Let me show you a few examples of what happens "under the hood", for your information. I'll use these parameters: **Interval = 1, Max False Positives = 5, Cool Down = 30**

Ex. 1: Application works without speed hacks. The detector runs its internal checks every second (**Interval** == 1). If something is wrong and timers desynchronize, the false positives counter will be increased by 1. After 30 seconds (**Interval** * **Cool Down**) of smooth work (without any OS hiccups) the false positives counter sets back to 0. It prevents undesired detection.

Ex. 2: Application started, and after some time, Speed Hack applied to the application. The detector runs its internal checks every second, raising the false positives counter by 1. After 5 seconds (**Interval** * **Max False Positives**) cheat will be detected.

If a cheater tries to avoid detection and will stop speed hack after 3 seconds, he has to wait for another 30 seconds before applying cheat again. Pretty annoying. And may be annoying even more if you increase Cool Down up to 60 (1 minute to wait). And cheaters have to know about Cool Down at all to make use of it.

SpeedHackProofTime APIs

With [SpeedHackDetector](#), you can utilize [SpeedHackProofTime](#) class to use reliable timers instead of Unity's [Time](#).* timers, which usually do suffer from the speed hacks. Mimics [Time](#).* APIs, except fixed* physics APIs. Works only when [SpeedHackDetector](#) is running and falls back to the Unity's timers until actual speed hack is detected. Use to make your animations and internal time-related calculations immune to the detected speed hack.

See usage example at the "[Examples/API Examples/Scripts/Runtime/InfiniteRotatorReliable](#)" script and at the "[API Examples](#)" scene.

IMPORTANT:

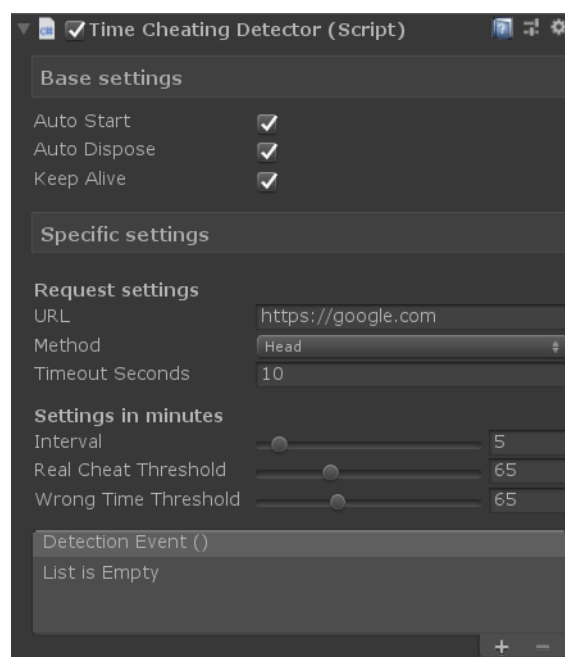
- Please note, [SpeedHackProofTime](#) may not work in some cases, e.g., when there is no legal way to reach reliable timers (may happen in sandbox processes, like processes in Chrome browser).

Time cheating detection

Players often use this type of cheating to accelerate some long-term processes in the game, like building progress, or energy cool down \ restore through a few hours or days. All cheater needs to do to leverage such cheating – just change system time to make your game "think" a day or few hours passed since he started to build a new item.

Anti-Cheat Toolkit's **Time Cheating Detector** allows detecting such cheating using online timeservers. You need an Internet connection to be able to use this detector.

See **Common detectors features and setup** topic above for the details on setup and common features of any detector.



URL: absolute URL to the resource which will return the correct Date response header value to the HEAD or GET request. When running in WebGL, the URL automatically changed to the current domain, if necessary, to avoid CORS limitations.

Method: request method to use. Head is faster and uses less traffic, but some web servers may treat too often HEAD requests as a bot activity and temporarily block the client. In case of any problems, use the Get method as it's more compatible.

Timeout Seconds: how long to wait for the server reply before aborting the request.

Interval: detection period (in **minutes**). Try to avoid using too short intervals (below 1 minute) to reduce traffic and extra resources' usage.

Real Cheat Threshold: Maximum allowed difference between two subsequent measurements of the online and offline time differences, in **minutes**. Actual cheat registered when the difference overshoots this value.

Wrong Time Threshold: Maximum allowed difference between online and offline time. When the difference overshoots this value, a callback or event raised (you can subscribe to these from scripting only) with check result:

[TimeCheatingDetector.CheckResult.WrongTimeDetected](#)

IMPORTANT:

- You may subscribe to the additional event: **CheatChecked** to get the full information about cheating check result. Same delegate is used for all the callbacks you can set through the scripting when starting detection.
- If there will be no Internet while checking for cheating, **CheatChecked** event will be raised with [CheckResult.Error](#) *result* argument and [ErrorKind.OnlineTimeError](#) *error* argument, check will just be skipped and scheduled for retry in specified interval again. The detector should work fine on devices with poor or time-to-time Internet connection.
- Since this detector needs the Internet, obviously it will generate additional permissions requests on mobile platforms, like [INTERNET](#) permission on Android (which is usually auto-granted). If you don't need this detector and wish to get rid of extra permission, please check the [ACTK_PREVENT_INTERNET_PERMISSION](#) in the [ACTk Settings](#) window.
- Any manual system time changes leading to difference between local UTC time and online UTC time will NOT be treated as a cheating, but will be treated as wrong time instead.
- You may use ForceCheck* methods to manually execute cheating check and get result (see API docs of these methods for usage examples). May be combined with 0 interval to switch the detector into the fully manual mode, when you manually execute checks at the desired moments.

Wallhacks detection [[video tutorial](#)]

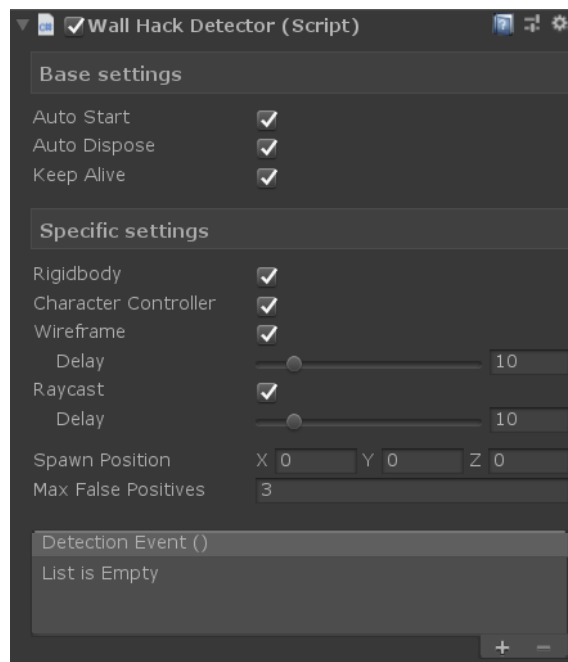
There are few different cheating methods which are often mentioned as "wall hack" – player can see through the transparent or wireframe walls, can walk through the walls like a ghost and can shoot through the walls.

ACTk has **WallHack Detector** which covers all of these.

It consists of few different modules, each detects different kind of cheats.

While running, WallHack Detector creates a service container "[WH Detector Service]" in the scene and uses it as a virtual sandbox within a 3x3x3 cube where it creates different items depending on your settings.

See **Common detectors features and setup** topic above for the details on setup and common features of any detector.



Rigidbody: enable this module to check for the "walk through the walls" kind of cheats made via Rigidbody hacks. Disable to save some resources if you're not using Rigidbody for characters.

Character Controller: enable this module to check for the "walk through the walls" kind of cheats made via Character Controller hacks. Disable to save some resources if you're not using Character Controllers.

Wireframe: enable this module to check for the "see through the walls" kind of cheats made via shader or driver hacks (wall became wireframe, alpha transparent, etc.). Disable to save some resources in case you aren't concerned about such cheats.

Note: requires specific shader to properly work at runtime. Read more details below.

Wireframe Delay: time in seconds between Wireframe module checks, from 1 up to 60 secs.

Raycast: enable this module to check for the "shoot through the walls" kind of cheats made via Raycast hacks. Disable to save some resources in case you aren't concerned about such cheats.

Raycast Delay: time in seconds between Raycast module checks, from 1 up to 60 secs.

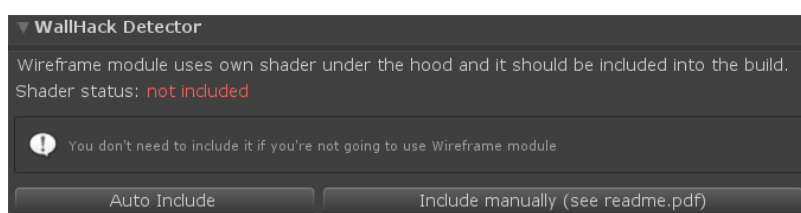
Spawn Position: world coordinates of the dynamic service container represented by a 3x3x3 red wireframe cube in the scene (visible when you select Game Object with detector).

Max False Positives: allowed detections in a row before actual detection. Each module has the own detection counter. It increases on every cheat detection and resets on the first success shot of the appropriate module (when a cheat is not detected). If any of such counters became more than Max False Positives value, the detector registers final, actual detection.

Wireframe module shader setup

The Wireframe module uses **Hidden/ACTk/WallHackTexture** shader under the hood. Thus, such a shader should be included into the build to exist at runtime. You'll see an error in logs at runtime if you'll have no **Hidden/ACTk/WallHackTexture** shader included, and you'll be prompted in the Editor to include it when you run WallhackDetector without shader included.

You may easily add or remove shader via the [ACTk Settings](#) window.

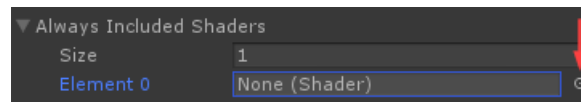


Press the "Auto include" button to automatically add the **Hidden/ACTk/WallHackTexture** shader to the [Always Included Shaders list](#).

You also may press the second button ("Include manually") to open Graphics Settings for your project and add the shader to the Always Included Shaders list manually.

To manually add shader to the list:

- add one more element to the list;
- click on the bulb next to the new empty element (see a red arrow on the image below);

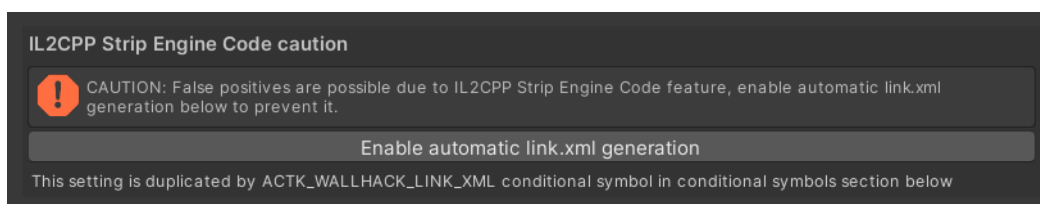


- search for "wallhack" in the opened window and select the **WallHackTexture** with double-click;

That's it for the wireframe module setup.

WallHack Detector and IL2CPP notice

When building with IL2CPP, Unity Engine stripping kicks in and removes unused components, which can occasionally remove stuff used by WallHack Detector, leading to false positives. To avoid this, make sure to enable automatic link.xml generation in the [ACTk Settings](#) (under WallHack Detector section):



IMPORTANT:

- Wallhacks are pretty specific kind of cheats usually applied to desktop FPS games, so make sure your game can suffer from them before using a detector, since detection requires a significant amount of extra resources.
- You may enable the **ACTK_WALLHACK_DEBUG** conditional compilation symbol in the [ACTk Settings](#) to keep the renderers on the service objects and see them in your game. It also enables OnGUI output of the resulting texture of the Wireframe module. This symbol works only in the development build or Editor.
- All objects within the service container are placed on the "Ignore Raycast" layer and have disabled renderers by default.
- It's important to set Spawn Position to the empty and isolated space to avoid any collisions of the 3x3x3 service sandbox with your objects, leading to the false positives and your objects' misbehavior.
- It's also essential to keep in mind detector may constantly create and move objects, use physics and make other calculations in the sandbox while running, depending on settings.

Managed DLL Injection detection [\[video tutorial\]](#)

IMPORTANT #1: Works only on Mono Android and Mono PC builds!

IMPORTANT #2: There is no assembly injection possible in IL2CPP (native injections are still possible). Read more at [code protection section](#).

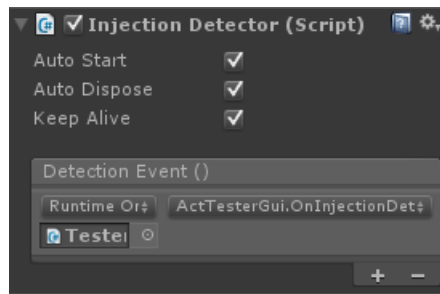
IMPORTANT #3: Disabled in Editor because of specific assemblies causing false positives. Use **ACTK_INJECTION_DEBUG** symbol to force it in the Editor.

WARNING: Please, test your app on the target device and make sure you have no false positives from the Injection Detector. If you have such an issue, try to add the detected assembly to the whitelist (covered below).

This kind of cheating is not as popular as others, though still used against Unity apps, so it can't be ignored. To implement such an injection, a cheater needs some advanced skills, thus many of them just buy cheat injectors from skilled people on special portals. One of the easiest ways to inject something – use [mono-assembly-injector](#) or similar software. Some nuts guys even use Cheat Engine's Auto Assemble for that. Do you see how cool the Cheat Engine is? 😊

Anti-Cheat Toolkit's **Injection Detector** allows reacting on injection of any *managed* assemblies (DLLs) into your app. Before using it in runtime, you need to enable it in [ACTk Settings](#) (Add mono injection detector support checkbox).

See **Common detectors features and setup** topic above for the details on setup and common features of any detector.



This detector has no additional options to tune except the common ones (note, when started from code, you pass a callback which accepts string argument – detection cause).
Simple way to detect advanced cheat!

Injection Detector internals (advanced)

Let me tell you a few words on the important Injection Detector "under the hood" topic to give you some insight. Usually, any Unity app may use three groups of assemblies:

- System assemblies: System.Core, mscorlib, UnityEngine, etc.
- User assemblies: any assemblies you may use in project, including third-party ones, like DOTween.dll (assembly from the great [DOTween](#) tweening library) or assemblies from the external packages + assemblies Unity generated from your code – Assembly-CSharp.dll, Assembly Definition assemblies, etc.
- Runtime generated and external assemblies. Some assemblies (such as [Microsoft.GeneratedCode](#)) could be created at runtime using reflection or loaded from external sources (e.g., web server) and you'll need to whitelist them manually. This is usually a rare case.

Injection Detector needs to know about all valid assemblies you trust to detect any foreign assemblies in your app. It automatically covers the first two groups. The last group should be covered manually (read below). Detector leverages a whitelist approach to skip allowed assemblies. It generates such a list automatically when you build your application and includes it into the build (if you have the [Enable Injection Detector](#) checkbox checked in the ACTk Settings window).

This whitelist consists of two parts:

- Dynamic whitelist from assemblies used in project (covers first and second group)
- User-defined whitelist (third group, see details below).

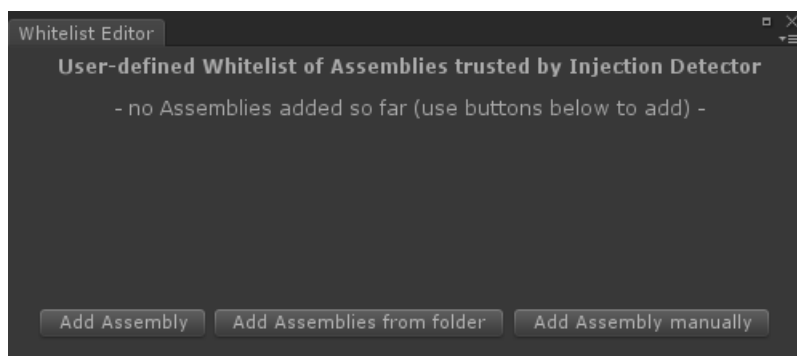
In most cases, you shouldn't do anything but just enable Injection Detector support in settings and set it up in scene \ code to let it work properly.

However, if your project loads some assemblies from external sources (and these assemblies are not present in your project assets) or generates assemblies at runtime, you need to let the detector know about such assemblies to avoid any false positives. For this reason, a user-defined whitelist editor was implemented.

How to fill user-defined whitelist

To add any assembly to the whitelist, follow these simple steps:

- Open Whitelist Editor using "[Tools > Code Stage > Anti-Cheat Toolkit > Injection Detector Whitelist Editor](#)" menu or "[Edit Whitelist](#)" button in the Settings window:



- Add new assembly using three possible options: single file ("Add Assembly"), recursive folder scan ("Add Assemblies from folder"), manual addition – filling up full assembly name ("Add Assembly manually").

That's it!

If you wish to add detected assembly manually and don't know its full name, just enable debug in the Injection Detector: check the `ACTK_INJECTION_DEBUG` in the [ACTk Settings](#) window and let the detector catch your assembly. You'll see its full name in the console log, after "[ACTk] Injected Assembly found:" string. This symbol works only in the development build or Editor.

Whitelist editor allows removing assemblies from the list one by one (with a small "-" button next to the assembly names) and clear the entire whitelist as well.

The user-defined whitelist is stored in the `ProjectSettings/ACTkSettings.asset` file with all other settings.

IMPORTANT:

- Please email me (see support contacts at the end of this document) your Invoice ID for injection example if you wish to try [InjectionDetector](#) in the wild.
- You may check the `ACTK_INJECTION_DEBUG_VERBOSE` and `ACTK_INJECTION_DEBUG_PARANOID` options in the [ACTk Settings](#) window to enable such compilation symbols for additional debug information. These symbols work only in the development build or Editor.

Third-party plugins integrations and notes

IMPORTANT:

Some third-party plugins with ACTk support may require `ACTK_IS_HERE` conditional symbol in your project. You can enable it in [ACTk Settings](#).

Obfuscator

The Anti-Cheat Toolkit provides different ways to prevent and detect cheating.

In addition, I strongly suggest complementing ACTk with good code protection to make your setup complete. Here are two first steps I suggest to make to increase the difficulty of your code reverse-engineering:

- Consider using IL2CPP instead of Mono, if possible, since this will make it harder for cheaters to inspect and analyze your code. IL2CPP builds do not have IL byte code, which cheaters usually decompile in ILSpy and such. Though cheaters still have a metadata of all your code (namespaces, class names, method names, and such) which could be used for bad. That's why the second step exists.
- Consider using an obfuscator to make class names, methods, fields and such a meaningless mess for the cheater. It will dramatically increase the difficulty of your code reverse-engineering. Make sure an obfuscator of your choice do support Unity IL2CPP.

Mfuscator

To make your build even more bullet-proof against reverse-engineering threat and prevent one-click hacking tools usage, I'd strongly recommend adding an extra layer of the native protection using [Mfuscator](#). It will encrypt IL2CPP metadata and introduce advanced difficulties to the hacker. They even have top-notch AAA-grade protector [available](#) for the Enterprise clients \o/

PlayMaker

Currently, ACTk has partial support for the [PlayMaker](#) (PM). There are few PM actions available in the package:

[Integration/PlayMaker.unitypackage](#). Import it into your project to add new actions to the PM Actions Browser. See a few examples in the [Scripts/PlayMaker/Examples](#) folder.

[ObscuredPrefs](#) and [ObscuredFilePrefs](#) are fully supported. You'll find [Obscured *](#) actions in the [PlayerPrefs](#) section of the Actions Browser.

- All detectors except the [ObscuredCheatingDetector](#) are fully supported. You'll find detectors actions in the [Anti-Cheat Toolkit](#) section of the Actions Browser.
- Basic Obscured types for PM are not currently supported. PM doesn't allow adding new variables types. But it's possible to integrate them by hands. There is an [example](#) and [explanation](#) by kreischweide.

Behavior Designer

Currently, ACTk has **full** support for the [Opsive Behavior Designer](#) (BD). There are few BD tasks (Actions & Conditionals) and SharedVariables available in the package:

[Integration/BehaviorDesigner.unitypackage](#). Import it into your project to integrate Anti-Cheat Toolkit with BD. See a few examples in the [Scripts/BehaviorDesigner/Examples](#) folder.

Other third-party plugins which play with ACTk nicely

- [Lovatto Studio](#) MFPS [Anti-Cheat And Reporting](#) addon.
- [Simple IAP System](#) and [IAP Receipt Validator](#) by [FLOBUK](#) to handle and validate your IAP receipts
- [Cross-Platform Native Plugins](#) by [Voxel Busters Interactive](#) to save into the cloud, setup leaderboards and more
- [Stan's](#) [Android Native Plugin](#).

Please let me know if you wish to see your plugin here.

Troubleshooting

- **I have errors after update.**
To avoid any update issues, please completely remove the previous version (whole [CodeStage/AntiCheatToolkit](#) folder) before importing the updated ACTk package into your project.
- **I have false positives or other misbehavior for one of the detectors.**
If you're starting your [existing in scene](#) detectors through the 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` (since it has callback with cause) will print details about detection if you'll enable [ACTK_DETECTION_BACKLOGS](#) flag in [ACTk Settings](#) window. Please report any false positives with logs generated on development build with this flag enabled if possible.
- **I have [InjectionDetector](#) 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](#)). For how to fill this whitelist, see [How to fill user-defined whitelist](#) section above. Contact me if it doesn't help.
- **I have [WallHackDetector](#) 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 the [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.
- **I have [ObscuredCheatingDetector](#) false positives.**
Please ensure you have no corrupted instances of the serialized obscured types. To do so, please navigate to the following menu: [Tools > Code Stage > Anti-Cheat Toolkit > Validate](#)
- **I'm having 'Can't add component because class * doesn't exist' error from [WallHackDetector](#).**
[WallHackDetector](#) does use these Unity components: `BoxCollider`, `MeshCollider`, `CapsuleCollider`, `Camera`, `Rigidbody`,

CharacterController, MeshRenderer. Such an error may appear when one of those components gets stripped out, e.g., by IL2CPP [strip engine code](#) option. To prevent components stripping, you can add them to any game object in any scene you have in your build or put such link.xml file anywhere in your Assets folder:

```
<linker>
  <assembly fullname="UnityEngine">
    <type fullname="UnityEngine.BoxCollider" preserve="all"/>
    <type fullname="UnityEngine.MeshCollider" preserve="all"/>
    <type fullname="UnityEngine.CapsuleCollider" preserve="all"/>
    <type fullname="UnityEngine.Camera" preserve="all"/>
    <type fullname="UnityEngine.Rigidbody" preserve="all"/>
    <type fullname="UnityEngine.MeshRenderer" preserve="all"/>
    <type fullname="UnityEngine.CharacterController" preserve="all"/>
  </assembly>
</linker>
```

ACTk tries to automatically handle this case and detect it at Play Mode to warn you and open settings where you can enable automatic link.xml file generation (using ACTK_WALLHACK_LINK_XML conditional compilation symbol).

- **How can I remove [READ_PHONE_STATE](#) permission requirement on Android?**

If you don't use the Device Lock feature in [ObscuredPrefs](#) / [ObscuredFile](#) / [ObscuredFilePrefs](#) and building an android application, I'd suggest checking the [ACTK_PREVENT_READ_PHONE_STATE](#) in the [ACTk Settings](#) window to remove [READ_PHONE_STATE](#) permission requirement on Android. Otherwise, you'll need this requirement to let ACTk lock your saves to the device.

- **How can I remove the [INTERNET](#) permission requirement on Android?**

If you don't use [TimeCheatingDetector](#), I'd suggest checking the [ACTK_PREVENT_INTERNET_PERMISSION](#) in the [ACTk Settings](#) window to remove the [INTERNET](#) permission requirement on Android. Otherwise, you'll need this requirement to let ACTk detect time cheating.

- **My obfuscator breaks Unity build when I'm using ACTk.**

If you're using some obfuscator which is not aware of Unity's Messages, Invoke()'s and Coroutines which may call methods by name, and your obfuscator is compatible with [System.Reflection.ObfuscationAttribute](#) attribute, feel free to check [ACTK_EXCLUDE_OBFUSCATION](#) in the [ACTk Settings](#) window to add the [\[Obfuscation\(Exclude = true\)\]](#) attribute to such methods. Contact me if it doesn't help.

- **I've updated ACTk from some old version, and now it can't read my ObscuredPrefs.**

If you're using [ObscuredPrefs](#) and wish to update ACTk, please consider the following: ACTk >= 1.4.0 can't decrypt data saved with ObscuredPrefs in ACTk < 1.2.5, so make sure you're not jumping from version < 1.2.5 directly to the version >= 1.4.0. You need to use any version in between to automatically convert prefs to the new format or get decryption code and bring it as an additional fallback to the ACT >= 1.4.0. In case of any issues with migration, feel free to contact me, and I'll help you make it smooth.

- **I'm having merge conflicts for the [Assets/Resources/fndid.bytes](#) file.**

If you're using version control for your project, and you're using [InjectionDetector](#), you may have merge conflicts. It's fine, just add that file to the ignore list of your VCS; [fndid.bytes](#) is automatically generated and can be different on the different machines.

- **I see StackOverflowException error in the Console.**

If you see such a message:

StackOverflowException: The requested operation caused a stack overflow.

CodeStage.AntiCheat.ObscuredTypes.ObscuredPrefs.HasKey (System.String key) (at

**CodeStage/AntiCheatToolkit/Scripts/ObscuredTypes/ObscuredPrefs.cs:*)*

More likely, you accidentally replaced all PlayerPrefs.* calls not only in your classes, but also in the ACTk classes as well. Just remove ACTk from your project and re-import it again and the issue should go away.

- **I see truncated or too short strings as a result of the [ObscuredString.GetEncrypted\(\)](#) \ [EncryptDecrypt\(\)](#) method(s).**

Obscured string may contain special characters, like line break, line end, etc. while encrypted. So, it may look broken, but its length and actual content should still be fine.

- **I see corrupted Obscured types after updating ACTk.**

Sometimes new ACTk versions have the vulnerability fixes leading to the serialization data model change which may render some types incorrectly after updating from older version, but in most cases it's possible to fix that automatically. There are few options here:

- menu commands at: [Tools > Code Stage > Anti-Cheat Toolkit > Migrate *](#) or [Validate *](#)
- context menu commands at Project View: [Code Stage > Anti-Cheat Toolkit > *](#)
- static APIs at obscured types for encrypted data from older versions: [MigrateEncrypted\(\)](#) or [DecryptFromV0\(\)](#)

If you are updating from 1.5.1.0 or earlier, you'll need to make an additional migration step, which was added in 1.5.2.0 after the first data model change. Please request older ACTk version to be able to make a full migration.

Or you can just reset broken variable in inspector and re-set it manually if you don't wish to migrate.

- I see **Error response code: 0** or **java.io.EOFException** errors in logs from **Time Cheating Detector** on old Android device (before Android 6).
There is a known bug in UnityWebRequest which reproduces only on old Android devices. Try setting the request method to GET instead of HEAD to work this bug around.
- I see **java.lang.ClassNotFoundException: net.codestage.*** errors in adb logs.
Make sure to exclude native ACTk Android plugin from obfuscation or minification. See [ProGuard notice](#) for more details.
- I can't read or write **ObscuredFile** or **ObscuredFilePrefs** from the **StreamingAssets** on Android or WebGL. This is an expected behavior due to the StreamingAssets nature at these platforms (see [Unity Manual](#) for more details). To work this around, please copy files from StreamingAssets folder to File.IO-accessible location (such as [Application.dataPath](#)) before reading them with ObscuredFile, like in [this simple example](#) made to demonstrate the idea.
- My CI environment (like GitHub actions) doesn't properly wait for the build completion and I'm not getting [CodeHashGeneratorPostprocessor.HashesGenerated](#) event.
In such case you can force synchronous hash generation using [CodeHashGeneratorPostprocessor.CalculateExternalBuildHashes\(buildPath, printToConsole\);](#) method call instead of subscribing to the [HashesGenerated](#) event.

Migration notes

Migrating from Anti-Cheat Toolkit v2021

When updating from ACTk v2021 you may need to update your code in order to use new APIs instead of deprecated ones, such as:

- There is no [CodeHashGeneratorPostprocessor.Instance](#) property available anymore since now you can reach all needed properties directly:
 - [Instance.callbackOrder](#) => [CodeHashGeneratorPostprocessor.CallbackOrder](#)
 - [Instance.HashesGenerated](#) => [CodeHashGeneratorPostprocessor.HashesGenerated](#)
- [HashGenerated](#) event handler signature changed – the [hashedBuilds](#) argument has a new type:
 - [BuildHashes\[\] hashedBuilds](#) => [IReadOnlyList<BuildHashes> hashedBuilds](#)
 - [BuildHashes.FileHashes](#) and [HashGeneratorResult.FileHashes](#) properties type changed from [Array](#) to [IReadOnlyList](#):
- [FileHash\[\] FileHashes](#) => [IReadOnlyList<FileHash> FileHashes](#)

Compatibility

All features with a few exceptions listed below should be fully functional on any known platform.

- [InjectionDetector](#) works only on Android Mono and Standalone Mono builds
- [CodeHashGenerator](#) works only on Android and Windows Standalone builds
- [AppInstallationSourceValidator](#) works only on Android platform

The plugin was tested on these platforms: **Standalone** (Win, Mac, Linux, WebGL), **iOS**, **Android**, **UWP**.

In addition, customers reported it as working on these:

Windows Phone 8, Apple TV (thx [atmuc](#)).

Please let me know if a plugin doesn't work for you on some specific platform, and I'll try to help with it.

Apple Encryption Export Regulations compatibility

All ACTk features are compatible with Apple's export compliance and U.S. export regulations and doesn't require setting `ITSAAppUsesNonExemptEncryption` plist.info key to YES.

These ACTk features are using exempt ("mass market") encryption by default:

- ObscuredFile and ObscuredFilePrefs
- ObscuredLong, ObscuredULong, ObscuredDouble and ObscuredDecimal types

The "mass market" encryption exempts from the export regulation according to United States Export Administration Regulations (EAR) Section [742.15](#). If you're using mentioned ACTk features in your app, you'll need to declare you're using encryption in your app when publishing to the Apple App Store, but this is an exempt encryption, not requiring licensing.

However, as stated in [Apple Documentation](#), you still may need to be required to submit a self-classification to U.S. government if you'll use the exempt encryption.

If you don't use any other exempt encryption in your app (like `https` / `tls` / `rsa` / `ssh` etc.) and suffer from ACTk requiring you to declare your app does uses encryption, you can prevent this with `ACTK_US_EXPORT_COMPATIBLE` option in the Conditional Compilation Symbols settings.

It does prevent generation of keys exceeding 56-bits for symmetric encryption, so it does not fall any more under the encryption definition at the Bureau of Industry and Security Commerce Control List's Category 5 Part 2, 2.a.1.a , so ACTk is not forcing you to declare you're using encryption in your application when you do publish it to the Apple App Store.

Please note, though, this comes with such side effects:

- ObscuredFile & ObscuredFilePrefs encryption strength weakening by switching from AES with 128-bit key to RC2 with 56-bit key (data will still be encrypted and not readable).
- Partial ObscuredLong, ObscuredULong, ObscuredDouble and ObscuredDecimal obscuration weakening, making it not fully encrypted in some rare cases. [ObscuredCheatingDetector](#) will keep an eye on them anyway.

Apple's Privacy Manifest

Anti-Cheat Toolkit usage does not require adding Privacy Manifest to your applications since it ACTk doesn't use any native [Required Reasons APIs](#) or [C# .Net framework APIs](#) which could trigger Apple Privacy Manifest requirement.

ProGuard notice

If you have minification enabled for Android build, make sure to exclude, i.e., add this to `proguard-user.txt`:

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

You can create `proguard-user.txt` file with these lines using **Tools > Code Stage > Anti-Cheat Toolkit > Configure proguard-user.txt** menu item.

Third party licenses included

xxHashSharp

<https://github.com/noricube/xxHashSharp>

BSD 2-Clause License (<http://www.opensource.org/licenses/bsd-license.php>)

SharpZipLib

<https://github.com/icsharpcode/SharpZipLib>

MIT License (<https://opensource.org/licenses/MIT>)

Final words from author

One more time - please keep in mind my toolkit is not something what can stop a very skilled, well-motivated cheater. There is no such solution actually, even giant anti cheat solutions, which cost thousands of dollars per month, still have flaws and cheating audience.

ACTk helps against most of the cheating players though, plus it will make advanced solo cheaters' life harder :P

I hope you will find **Anti-Cheat Toolkit** suitable for your needs, and it will save some of your priceless time! Please [leave your reviews](#) on the plugin's Asset Store page and please have no hesitation to drop me bug reports, feature suggestions and other thoughts on the forum or via support contacts you'll find below.

Thanks for taking your time to read this document!




Anti-Cheat Toolkit links and support

[Asset Store](#) | [Homepage](#) | [API](#) | [Changelog](#) | [Discussions](#) | [YouTube](#)

Support contacts:

discord.gg/KrU4psWffA
codestage.net/contacts
support@codestage.net

Subscribe for updates and news:

 [Discord Announcements Channel](#)
 [@codestage_net](#)
 [@codestage](#)

Best wishes,
Dmitry Yuhanov
[Asset Store publisher](#)
codestage.net

P.S. #0 I wish to thank my family for supporting me in my Unity Asset Store efforts and making me happy every day!
P.S. #1 I wish to say huge thanks to [Daniele Giardini](#) ([DOTween](#), [HOTools](#), [Goscurry](#) and much other happiness generating things creator) for awesome logos, intensive help and priceless feedback on this toolkit!