Hey there!

Huge thanks for buying **Anti-Cheat Toolkit (ACTk)**, multi-purpose anti-cheat solution for Unity3D!
Current version handles these types of cheating:

- **Memory cheating** (prevention + detection)
- **Saves cheating** (prevention + detection)
- **Speed hack** (detection)
- **Wall hack** (detection)
- **DLL injection** (detection)

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 are able to break anything!

Full API documentation can be found here:
http://codestage.ru/unity/anti-cheat/api/
Please, consider visiting it first if you have any questions on plugin usage.

Plugin features in-depth

Memory cheating (introduction video)

This is 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:
ObscuredFloat
ObscuredInt
ObscuredString
ObscuredVector3
and more (all basic types + few Unity-specific types are covered)...

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

```
// place this line right at the beginning of your .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);
```

Easy, right? 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 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 detect their efforts. Actual obscured values are still safe in such case: 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.

Few more words about obscured types

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

Obscured types pro-tips:

- All simple Obscured types have public static methods **GetEnctypted()** and **SetEncrypted()** to let you work with encrypted value from obscured instance. May be helpful if you're going to use some custom saves engine.
- You may (and should!) change default encryption keys in all obscured types (including ObscuredPrefs described below) using public static function **SetNewCryptoKey()**. Please, keep in mind, all new obscured types instances will use new key, but all current instances will keep using previous key unless you explicitly call **ApplyNewCryptoKey()**.
- 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.

IMPORTANT:

- Currently these types can be exposed to the inspector: ObscuredString, ObscuredBool, ObscuredInt, ObscuredFloat, ObscuredLong, ObscuredDouble, ObscuredVector2, ObscuredVector3. Be careful while replacing regular types with the obscured ones – inspector values will reset!
- Obscured types are usually require additional resources comparing to the regular ones. Please, try keeping obscured variables away from Updates, loops, huge arrays, etc. and keep an eye on your Profiler, especially while working on mobile projects.
- LINQ is not supported at this moment.
- XmlSerializer is not supported at this moment.
- Binary serialization is supported out of the box, JSON is also possible using ISerializable implementation (example).
- Generally, I'd 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.

Saves cheating (introduction video)

Now it's time to speak about **saves** cheating a bit. Unity developers often use PlayerPrefs (PP) class to save some in-game data, like player game progress or in-game goods purchase status. However, not all of us know how easily all that data 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 cheatersproof! ☺

Here is a simple example:

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

ObscuredPrefs.SetFloat("currentLifeBarObscured", 88.4f);
float currentLifeBar = ObscuredPrefs.GetFloat("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 old good regular PlayerPrefs class, but now your saves are secure from cheaters (well, from most of them)!

ObscuredPrefs has some additional functionality:

- You may subscribe to the **onAlterationDetected** callback. It allows you to know about saved data alteration. Callback will fire once (for whole session) as soon as you read some altered data.
- **lockToDevice** field 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 persistent device ID on iOS. So avoid using it or use in conjunction with **DeviceId** property 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 ObscuredPrefs.DeviceLockLevel and **lockToDevice** descriptions in API docs for more info.
  Android users: see **Troubleshooting** section below if you don't use this feature.
- You may subscribe to the **onPossibleForeignSavesDetected** callback if you use the **lockToDevice** field. It allows you to know if saved data are from another device. It's checked on data read, so it will fire once (per session) as you read first suspicious data piece. **Please note:** callback may fire also if cheater tried to modify some specific parts of saved data.
- You may restore all locked to device data in emergency cases (like device ID change) using **emergencyMode** flag. Set it to true to decrypt any data (even locked to another device).
- By default, saves from other devices are not readable. But you may force OP to read them using **readForeignSaves** field. **onPossibleForeignSavesDetected** callback still will be fired**.**
- OP supports some additional data types comparing to regular PP: uint, double, long, bool, byte[], Vector2, Vector3, Quaternion, Color, Rect.

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 original key will be kept, allowing you to read it again using PP. You may see **preservePlayerPrefs** in action in the TestScene.
- 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 huge portions of data like maps arrays, your own database, etc. - use regular disk IO operations for that.
- Use **ForceLockToDeviceInit()** to avoid possible gap on first data load / save with **lockToDevice** enabled. It may happen if unique device id obtaining process requires significant amount of time. You may use this method in such case to force this device id obtaining process to run at desired time, while you showing something static, like a splash screen.
- Use **DeviceId** property to explicitly set device id when using **lockToDevice**. 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 consistent device ID (on iOS 7+), all we have is Vendor and Advertisement IDs, both can change and have restrictions \ corner cases.
- 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.
- You may use **unobscuredMode** field to write all data unencrypted in editor, like regular PP. Use it for debugging, it works only in editor and breaks PP to OP migration (not sure you need it in editor anyway though).

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. Feel free to check it yourself, using Preformance Obscured Tests component on the PerformanceTests game object in the TestScene.

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 3 ways to setup and use any detector:

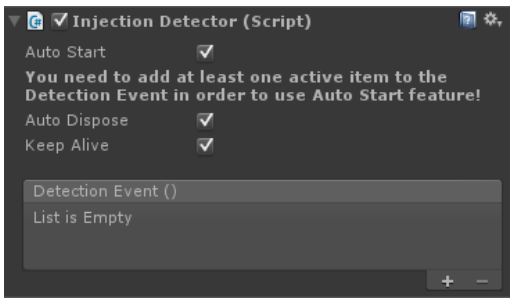
- setup through the Unity Editor
- setup through the code
- mixed mode

Setup through the Unity Editor

First of all, you need to add detector to the scene. You have 2 options for that:

1. Use GameObject > Create Other > Code Stage > Anti-Cheat Toolkit menu to quickly add detector to the scene. "Anti-Cheat Toolkit Detectors" Game Object will be created in scene (if it doesn't exist) with detector component attached to it.
   This is recommended way to setup detectors.
2. Alternatively, you may add 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 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). In order to use this feature you should configure *Detection Event* described below.

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

Keep Alive: enable to let detector survive new level (scene) load. Otherwise 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 will travel through the scenes together.

Setup through the code

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

If you wish to tune any options – just use 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 scene).

Some specific for detector options usually available for initial configuration through the arguments of the overloaded **StartDetection()** methods, see 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 your .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 setup them in a mixed way – combining configuration in the inspector and manual launch through the code.

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

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

In order to use such 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).

Other notes on all detectors

On dispose Detector follows 2 rules:

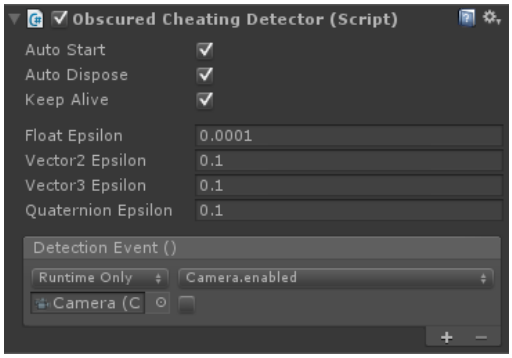
- If Detector is placed on the Game Object named "Anti-Cheat Toolkit Detectors": Game Object will be automatically destroyed if no other Detectors left attached regardless of any other components or children.
- If Detector is placed on the Game Object with any other name: Game Object will be automatically destroyed only if it has neither other components nor children attached.

Obscured types cheating detection

Anti-Cheat Toolkit's **Obscured Cheating Detector** allows to detect cheating of all obscured types except ObscuredPrefs (it has own detection mechanisms covered in 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.



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).

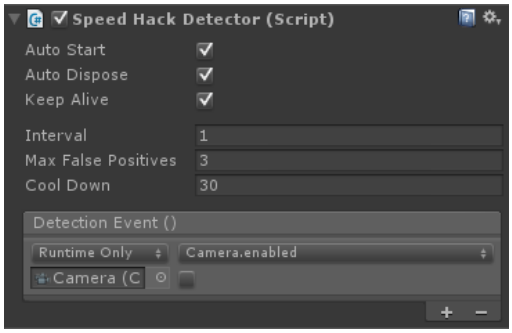
Note: cheating detection will not work and fake unencrypted variables will not exist in memory while detector is not running. So if you don't use ObscuredCheatingDetector or you have disabled Auto Run in inspector and you don't run it from code - your values will not appear in any memory searchers at all.

Speed hack detection (introduction video)

This type of cheating is very popular and used pretty often since it is really easy to use and any child around may try it on your game. Most popular tool for that – Cheat Engine. It has built-in Speed Hack feature allowing speeding up or slowing down target application with few simple clicks.

Anti-Cheat Toolkit's **Speed Hack Detector** allows to detect Cheat Engine's (and some other tools) speed hack usage. It's really easy to use as well ;)

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.

Max False Positives: in some very rare cases (system clock errors, OS glitches, etc.) detector may produce false positives, this value allows skipping specified amount of speed hack detections before reacting on cheat. When actual speed hack is applied – 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, detector will fire detection event in ~5 seconds after that (Interval * Max False Positives + time left until next check).

Cool Down: allows to reset internal false positives counter after specified amount of successful shots in the row.
It may be useful if your app have 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 value to 0 to completely disable cool down feature.

Let me show you 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. Detector runs its internal checks every second (**Interval** == 1). If something is wrong and timers desynchronize, false positives counter will be increased by 1. After 30 seconds (**Interval** * **Cool Down**) of smooth work (without any OS hiccups) 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. Detector runs its internal checks every second, raising false positives counter by 1. After 5 seconds (**Interval** * **Max False Positives**) cheat will be detected. If cheater will try to avoid detection and will stop speed hack after 3 seconds, he have 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 cheater have to know about Cool Down at all to try make use of it.

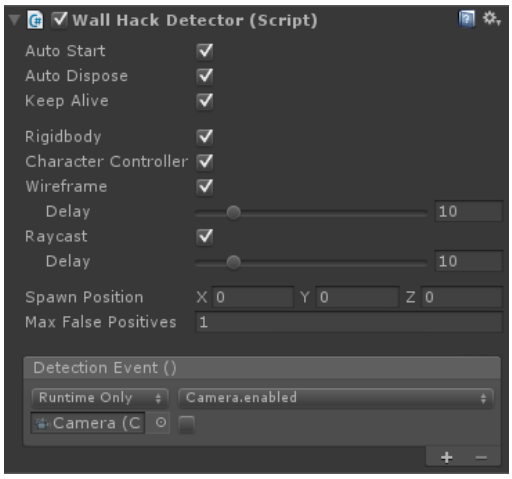
Wallhacks detection (introduction video)

There are few different cheating methods which are usually 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 service container "[WH Detector Service]" in scene and uses it as a virtual sandbox within 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 don't care about such cheats.
Note: requires specific shader in order 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 don't care 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 3x3x3 red wireframe cube in scene (visible when you select Game Object with detector).

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

Wireframe module shader setup

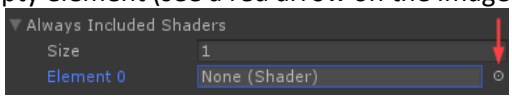
Wireframe module uses Hidden/ACTk/WallHackTexture shader under the hood. Thus such shader should be included into the build to exist at runtime. You'll see error in logs at runtime if you'll have no Hidden/ACTk/WallHackTexture shader included.

You may easily add or remove shader via the ACTk Settings window. Just use the
Window > Code Stage > Anti-Cheat Toolkit > Settings menu to open settings window.

Press the "Include Shader" button to automatically add the Hidden/ACTk/WallHackTexture shader to the Always Included Shaders list.

You also may press the second button ("Open Graphics Settings…") to open Graphics Settings for your project and add 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.

IMPORTANT:

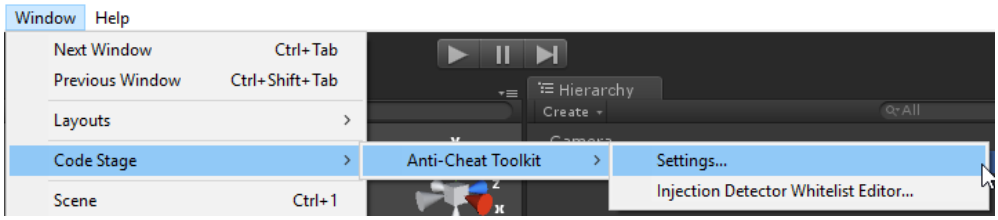
- Wallhacks are pretty specific kind of cheats usually applied to desktop FPS games, so make sure your game can suffer from them before using detector, since detection requires significant amount of extra resources.
- You may add the ACTK_WALLHACK_DEBUG conditional compilation symbol to the **Scripting Define Symbols** in the **Player Settings** (as described in the docs) 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 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 3x3x3 service sandbox with your objects leading to the false positives and your objects misbehavior.
- It's also important to keep in mind detector may constantly create and move objects, use physics and make other calculations in sandbox while running, depending on settings.

DLL Injection (introduction video)

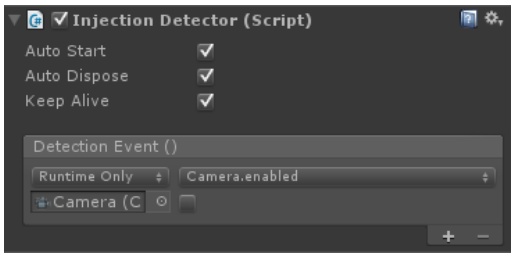
IMPORTANT #1: Works for Android and Standalone builds (including WebPlayer)!
IMPORTANT #2: Disabled in Editor because of specific assemblies causing false positives. Use ACTK_INJECTION_DEBUG symbol to force it in Editor.
WARNING: Please, test you app on target device and make sure you have no false positives from Injection Detector. If you have such issue try to add 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 injection, cheater needs some advanced skills, thus many of them just buy cheat injectors from skilled people on special portals. One of easiest way to inject something – use mono-assembly-injector. There are few other injectors used too. Some nuts guys even use Cheat Engine's Auto Assemble for that. Do you see how cool Cheat Engine is? ☺

Anti-Cheat Toolkit's **Injection Detector** allows to react on injection of any managed assemblies (DLLs) into your app.
Before using it in runtime, you need to enable it in Anti-Cheat Toolkit Settings in editor:
"Window > Code Stage > Anti-Cheat Toolkit > Settings…"



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.
Simple way to detect advanced cheat!

Injection Detector internals (advanced)

Let me tell you 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, mscoree, 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) + assemblies Unity generated from your code - Assembly-CSharp.dll, etc.
- Runtime generated and external assemblies. Some assemblies could be created at runtime using reflection or loaded from external sources (e.g. web server).

Injection Detector needs to know about all valid assemblies you trust to detect any foreign assemblies in your app.
It automatically covers first two groups. Last group should be covered manually (read below).

Detector leverages whitelist approach to skip allowed assemblies. It generates such list automatically while you work on your project in the Unity Editor and stores it in the Resources/fn.bytes file.

This whitelist consists of three parts:

- Contents of the Editor/ServiceData/InjectionDetectorData/DefaultWhitelist.bytes file with default static whitelist (covers first group).
- Dynamic whitelist from assemblies used in project (covers second group); it updates automatically after scripts compilation.
- User-defined whitelist (third group, see details below).

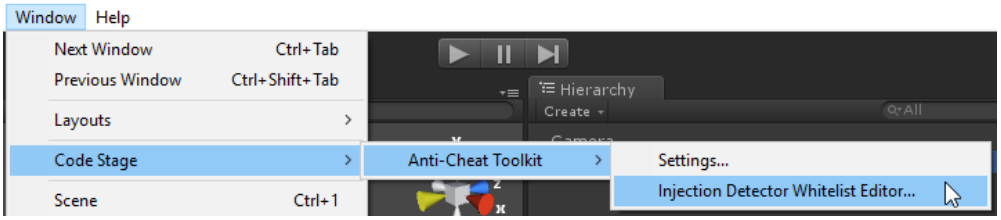
In most cases, you shouldn't do anything but just enabling Injection Detector in settings and setting it up in scene \ code to let it work properly.

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 detector know about such assemblies to avoid any false positives. For this reason user-defined whitelist editor was implemented.

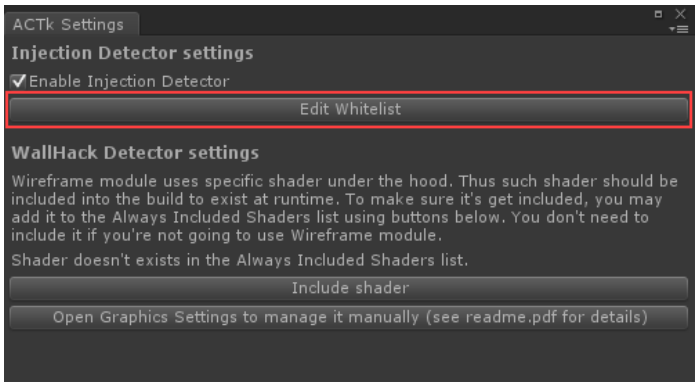
How to fill user-defined whitelist

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

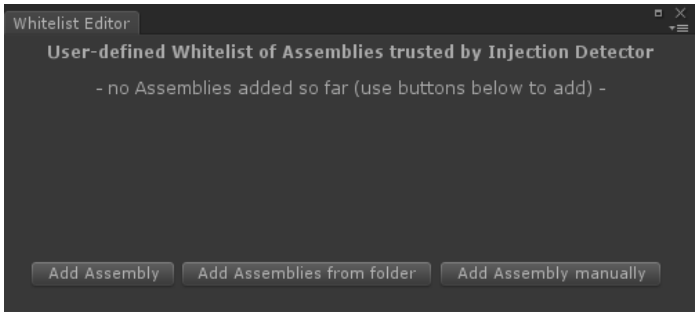
- Open Whitelist Editor using "Window > Code Stage > Anti-Cheat Toolkit > Injection Detector Whitelist Editor" menu or "Edit Whitelist" button in the Settings window:



or



opens this 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 assembly manually and don't know its full name, just enable debug in the Injection Detector (add the ACTK_INJECTION_DEBUG conditional compilation symbol to the **Scripting Define Symbols** in the **Player Settings** as described in the docs) and let detector catch your assembly. You'll see its full name in console log, after "[ACTk] Injected Assembly found:" string. This symbol works only in the development build or Editor.

Whitelist editor allows to remove assemblies from the list one by one (with small "-" button next to the assembly names) and clear entire whitelist as well.
User-defined whitelist is stored in the Editor/ServiceData/InjectionDetectorData/UserWhitelist.bytes file.

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 add the ACTK_INJECTION_DEBUG_VERBOSE and ACTK_INJECTION_DEBUG_PARANOID conditional compilation symbols to the **Scripting Define Symbols** in the **Player Settings** (as described in the docs) for additional debug information. These symbols work only in the development build or Editor.

Third party plugins

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 few examples at the Scripts/PlayMaker/Examples folder.

ObscuredPrefs is fully supported. You'll find Obscured Prefs * actions in the PlayerPrefs section of the Actions Browser.

- All detectors except the ObscuredCheatingDetector are fully supported. You'll find detectors actions at the Anti-Cheat Toolkit section of the Actions Browser.
- Basic Obscured types for PM are not currently supported. PM doesn't allow to add new variables types. But it's possible to integrate them by hands. There is 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 few examples at the Scripts/BehaviorDesigner/Examples folder.

Other third-party plugins with ACTk support

There are some other third-party plugins with ACTk support:

- **Mad Level Manager**: ACTk is used as storage backend.
- **Stan's Assets**: **Android Native** and **Ultimate Mobile** plugins.
- **UFPS**: ACTk is going to be used with multiplayer kit (no ETA, sorry).

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

Tips & Tricks

- Try to place your detectors to the Anti-Cheat Toolkit Detectors game object if possible. If all attached detectors self-destroy, it will be destroyed as well ignoring other attached scripts and nested objects (by default detector's object is not destroyed if it has another name and if it has children or other scripts). It allows you to attach to this object any scripts you wish to migrate from scene to scene with "KeepAlive" detectors and behave properly on returning to the initial scene.
- ACTk is compatible with Apple's export compliance by default since it doesn't use keys in excess of 56-bits for symmetric encryption by default, so it doesn't fall under the encryption definition at the Bureau of Industry and Security Commerce Control List's Category 5 Part 2, 2.a.1.a. Find more details on Unity forums. So if you'll use keys longer than 7 chars, you're about to fall under the EAR control and it may be necessary to declare that you are using encryption in certain countries. For example, when submitting an app to the Apple App Store.

Troubleshooting

- **I have errors after update.**
  To avoid any update issues completely remove previous version (whole CodeStage/AntiCheatToolkit folder) before importing updated ACTk package into your project.
- **I have false positives or other misbehaviour for one of 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.
- **I have InjectionDetector false positives.**
  Make sure you've added all external libraries your game uses to the whitelist (menu: Window > 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 helps.
- **I have WallHackDetector false positives**.
  Make sure you've properly configured **Spawn Position** of the detector and detector's service objects are not intersecting with any objects from your game. Also make sure 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 make sure Ignore Raycast layer will collide with itself.
- **How can I remove READ_PHONE_STATE permission requirement on Android?**
  If you don't use ObscuredPrefs.lockToDevice feature and building an android application, I'd suggest to add the ACTK_PREVENT_READ_PHONE_STATE conditional compilation symbol to the **Scripting Define Symbols** in the **Player Settings** (as described in the docs) to remove READ_PHONE_STATE permission requirement on Android. Otherwise, you'll need this requirement to let ACTk lock your saves to the device.
- **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 add the ACTK_EXCLUDE_OBFUSCATION conditional compilation symbol to the **Scripting Define Symbols** in the **Player Settings** (as described in the docs) to add [Obfuscation(Exclude = true)] attribute to such methods. Contact me if it doesn't help.
- **I've updated ACTk from some very old version and now it can't read my ObscuredPrefs.**
  If you're using ObscuredPrefs and wish to update ACTk, please consider 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 message:
  StackOverflowException: The requested operation caused a stack overflow.
  CodeStage.AntiCheat.ObscuredTypes.ObscuredPrefs.HasKey (System.String key) (at Assets/CodeStage/AntiCheatToolkit/Scripts/ObscuredTypes/ObscuredPrefs.cs:)*
  More likely you accidentally replaced all PlayerPrefs.* calls not only in your classes, but in the ACTk classes as well. Just remove ACTk from your project and re-import it again and issue should go away.

Compatibility

Plugin should work on any platform generally. Some features have platform limitations though; they are mentioned here and / or in API docs in such cases.

I had no chance to test plugin on all Unity platforms since I just have no appropriate devices and / or licenses. Plugin was tested on these platforms: **PC** (Win, Mac, WebPlayer, WebGL), **iOS**, **Android, Win Phone (Emulator), Windows Store**. Please, let me know if plugin doesn't work for you on some specific platform and I'll try to help and fix it.

All features should work fine with **micro mscorlib** stripping level, **.NET 2.0 Subset** API level and **IL2CPP**.
Plugin **doesn't require Unity Pro** license.

Plugin may work with JS code after some handwork on preparing it for such usage. Read more here (outdated):
http://forum.unity3d.com/threads/anti-cheat-toolkit-released.196578/page-3#post-1573781

Final words

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 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 anti-cheat needs and it will save some of your priceless time! Please, leave your reviews at the plugin's Asset Store page and feel free to drop me bug reports, feature suggestions and other thoughts on the forum or via support contacts!

Thanks for taking your time to read this document!

ACTk links:

Asset Store | Web Site | Forum | YouTube

Support contacts:

E-mail: focus@codestage.ru
Other: blog.codestage.ru/contacts

Best wishes,
Dmitriy Yukhanov
Asset Store publisher
blog.codestage.ru
@dmitriy_focus

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 many other happiness generating things creator) for awesome logos, intensive help and priceless feedback on this toolkit!*