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)
- **DLL injection** (detection, experimental)

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!

IMPORTANT:

New version usually needs some time to appear at Asset Store, but you may grab it from me directly – just drop me your Invoice ID and I'll send you ACTk update. You'll know about new version from my twitter, website or from Unity forums.

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.

Memory cheating (introduction tutorial)

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, heath, score, etc. Most popular tools: Cheat Engine (PC), Game CIH (Android). There are plenty of other tools for these and other platforms as well.

To leverage **memory** anti-cheat techniques I prepared for you bunch of obscured types to use instead of regular ones:

```
ObscuredFloat
ObscuredInt
ObscuredString
ObscuredVector3
and more (all basic 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.

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 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. Setup is common for all ACTk detectors:

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

// you may place this line once at Awake() or Start() handler of MonoBehaviour for example
// it makes few important steps under the hood:
// - adds detector component onto the Anti-Cheat Toolkit Detectors Game Object
//   - if Game Object doesn't exists it creates automatically
//   - if detector component already exists somewhere in scene, its instance used instead
// - starts detection engine with specified callback
ObscuredCheatingDetector.StartDetection(OnObscuredTypeCheatingDetected);
// ...

// this method will be called once on Obscured type cheating try detection
private void OnObscuredTypeCheatingDetected()
{
	Debug.Log("Gotcha, nasty cheater!");
}
```

Cheating detection works for **all** obscured types.
Note: ObscuredPrefs has own detection mechanisms covered below.

Cheating detection will not work and fake variables will not exist in memory until you call **StartDetection()** method. ObscuredCheatingDetector has epsilons for ObscuredFloat, ObscuredVector2, ObscuredVector3 and ObscuredQuaternion exposed to the inspector. Epsilon is a 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).

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

IMPORTANT:

- Currently only these types can be exposed to the inspector: ObscuredString, ObscuredBool, ObscuredInt and ObscuredFloat. Be careful while replacing regular types with the obscured ones – inspector values will reset!
- 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 about to use some custom saves engine.
- Obscured types 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 tutorial)

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 **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 **ForceLockToDeviceInit()** 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.
- You may subscribe to **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 such data using **readForeignSaves** field. **onPossibleForeignSavesDetected** callback still will be triggered**.**
- OP supports some additional data types comparing to regular PP: double, long, bool, byte[], Vector2, Vector3, Quaternion, Color.

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! 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 could use this method in such case to force this device id obtaining process to run at desired time, while you showing a splash screen for example.
- Use **ForceDeviceID()** 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.
- **You may (and should!) change default encryption keys in all obscured types (including ObscuredPrefs) using public static function SetNewCryptoKey(). Please, keep in mind, all new obscured types instances will use specified key, but all current instances will keep using previous key unless you explicitly call ApplyNewCryptoKey() on them.**

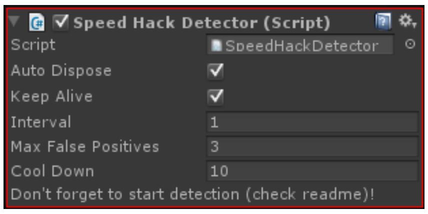
Speed hack (introduction tutorial)

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 ;)
Generally all you need to do - just call SpeedHackDetector.StartDetection() once anywhere in your code to get detector running with default parameters (or with parameters set in inspector). Simple, isn't it?

Detector has some parameters available for tuning from inspector or via **StartDetection()** method arguments. Generally, you may control detector entirely from code, without adding it to the scene, it will be added there automatically if it doesn't exists.

To set up detector in editor and set all parameters in inspector just add it to any Game Object as usual or use menu item "GameObject > Create Other > Code Stage > Anti-Cheat Toolkit > Speed Hack Detector".
"Anti-Cheat Toolkit Detectors" Game Object will be created in scene (if it doesn't exist) with component attached to it:


Speed Hack Detector (Script)

Script	SpeedHackDetector
Auto Dispose	✓
Keep Alive	✓
Interval	1
Max False Positives	3
Cool Down	10

Don't forget to start detection (check readme)!

This component can be placed on any Game Object.
Best practice is to use single Anti-Cheat Toolkit Detectors Game Object for all detectors.

Auto Dispose: Speed Hack Detector component will self-destroy after speed hack detection. Game Object will be automatically destroyed as well if no other components left on it and it has no children.

Keep Alive: Speed Hack Detector component will survive new level (scene) load if checked. Otherwise component will be destroyed. Game Object will be automatically destroyed as well if no other components left on it and it has no children.

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, 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 not detections in the row.
It may be useful if your app have rare yet periodic false detections (in case of some 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 "under the hood" processes 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 fires 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 is set back to 0. It prevents undesired detection.

Ex. 2: Application started, and after some time Speed Hack applied to the application. Detector fires 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.

Don't forget you still need to call SpeedHackDetector.StartDetection() from code to start detector (it's not enough to just add it to the scene):

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

// starts detector with specified callback
SpeedHackDetector.StartDetection(OnSpeedHackDetected);

// this method will be called as speed hack detected and confirmed
private void OnSpeedHackDetected()
{
	Debug.Log("Gotcha!");
}
```

In this simple example you may notice there is only one argument passed to the **StartDetection()** method – OnSpeedHackDetected callback. All other parameters (interval, max false positives and cool down) are either default (1, 3, 30, if you didn't add detector to the scene) or initialized from inspector values.

If you wish to have some additional control, you may pass few additional arguments into the **StartDetection()** method to override any default parameters, or parameters set in inspector:

```
SpeedHackDetector.StartDetection(OnSpeedHackDetected, 1f, 5, 60);
```

In this case there are 4 arguments passed to the **StartDetection()** method – OnSpeedHackDetected callback, 1 second detection interval, 5 allowed false positives and 60 "no cheat" cool down shots.

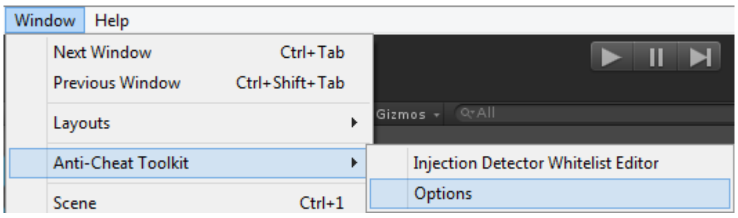
See the example of Speed Hack Detector set up in the TestScene and feel free to use prefab with it from "CodeStage\AntiCheatToolkit\Examples\Prefabs\Anti-Cheat Toolkit Detectors".

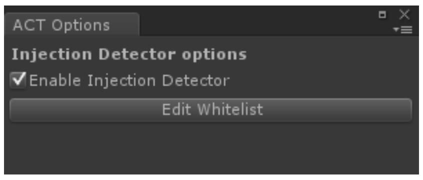
DLL Injection (introduction tutorial)

IMPORTANT #1: Works on Android, iOS and PC (including WebPlayer)!
IMPORTANT #2: Doesn't work in editor (since editor uses editor-specific assemblies).
WARNING: This feature is in experimental state ATM. Use it in production with extra caution! Please, test you app on target device and make sure you have no false positives from Injection Detector. If you have such issue – I'd be glad to get in touch with you and help to resolve it.

This kind of cheating is not as popular as others, though still used against Unity apps sometimes so it can't be ignored. To implement such injection, cheater needs some advanced skills, so many of them just buy injectors from skilled people on special portals. 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 Options in editor:
"Window > Anti-Cheat Toolkit > Options"




ACT Options
Injection Detector options
☑ Enable Injection Detector

Setup is similar to any other detector: generally all you need to do - just call InjectionDetector.StartDetection() once anywhere in your code:

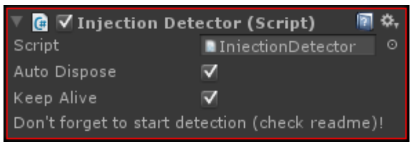
```
// place this line right in the beginning of your .cs file!
using CodeStage.AntiCheat.Detectors;

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

// this method will be called on injection detect
private void OnInjectionDetected()
{
	Debug.Log("Gotcha!");
}
```

Simple way to detect advanced cheat!

To set up detector in editor and set all parameters in inspector just add it to any Game Object as usual or use menu item "GameObject > Create Other > Code Stage > Anti-Cheat Toolkit > Injection Detector".
"Anti-Cheat Toolkit Detectors" Game Object will be created in scene (if it doesn't exist) with component attached to it:


Injection Detector (Script)
Script InjectionDetector
Auto Dispose ✓
Keep Alive ✓
Don't forget to start detection (check readme)!

This component can be placed on any Game Object.
Best practice is to use single Anti-Cheat Toolkit Detectors Game Object for all detectors.

Settings you see in inspector are equal to same settings from Speed Hack Detector:

Auto Dispose: Injection Detector will self-destroy after detection reporting. Game Object will be automatically destroyed as well if no other components left on it and it has no children.

Keep Alive: Injection Detector component will survive new level (scene) load if checked. Otherwise component will be destroyed. Game Object will be automatically destroyed as well if no other components left on it and it has no children.

Don't forget you still need to call InjectionDetector.StartDetection() from code to start detector!

See an example of the Injection Detector setup in the TestScene and feel free to use prefab with it from "CodeStage\AntiCheatToolkit\Examples\Prefabs\Anti-Cheat Toolkit Detectors".

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, 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 leverage whitelist approach used 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)

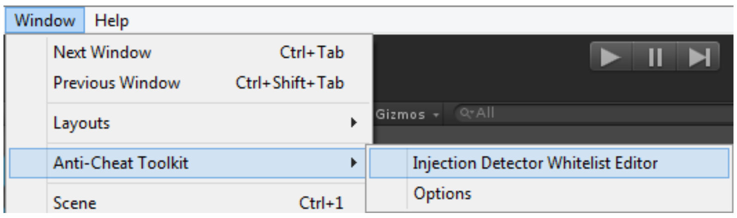
In most cases, you shouldn't do anything but just enabling Injection Detector in options and starting it in script to let it work properly.

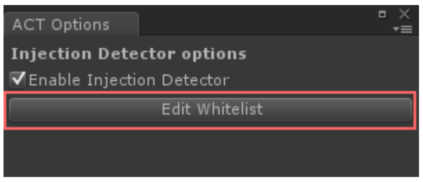
However, if your project loads some assemblies from external sources (and these assemblies are not present in your project assets) or generates assemblies at runtime, you need to let Detector know about such assemblies to avoid any false positives. For this reason user-defined whitelist editor was implemented.

How to fill user-defined whitelist

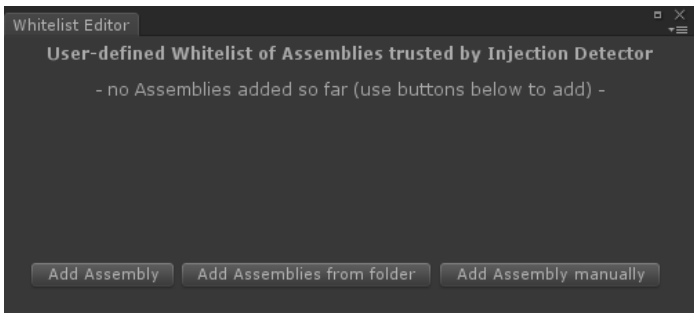
To add any assembly to the whitelist, follow these simple steps:

- open Whitelist Editor using "Window > Anti-Cheat Toolkit > Injection Detector Whitelist Editor" menu or "Edit Whitelist" button in the Options window:





✓ Enable Injection Detector



User-defined Whitelist of Assemblies trusted by Injection Detector

- no Assemblies added so far (use buttons below to add) -

- 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 (comment out #undef DEBUG line right in the beginning of the InjectionDetector.cs file) and let detector catch your assembly. You'll see its full name in console log, after "[ACTk] Injected Assembly found:" string.
Whitelist editor allows to remove assemblies from it 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.

Third party plugins support

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 in experimental mode (I need to receive a bit of feedback from PM users before claiming it stable). You'll find Obscured Prefs * actions in the PlayerPrefs section of the Actions Browser.

- SpeedHackDetector is fully supported in experimental mode (I need to receive a bit of feedback from PM users before claiming it stable). You'll find Speed Hack Detected action in 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.
- Other detectors are not supported yet (may change in future).

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.

Currently BD support should be considered as experimental (I need to receive a bit of feedback from BD users before claiming it stable).

Please, let me know if you wish to see some other third-party plugins here.

Troubleshooting

- To avoid any update issues: completely remove previous version (whole CodeStage/AntiCheatToolkit folder) before importing updated ACTk package into your project.
- If you have Injection Detector false positives: add all external libraries your game uses to the whitelist (menu: Window > Anti-CheatToolkit > Injection Detector Whitelist Editor). For how to fill this whitelist, see "***How to fill user-defined whitelist***" section above.

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 platforms Unity supports since I just have no appropriate devices and / or licenses.
Plugin was tested on these platforms: **PC** (Win, Mac, WebPlayer), **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 and **.NET 2.0 Subset** API level.
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 would not stop very skilled well-motivated cheater.
It will filter a lot of script-kiddies and cheaters-newbies though, plus it will make advanced 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!

Please consider visiting ACTk YouTube playlist to watch some tutorials and examples (and subscribe for updates!).

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!*