Hey there!

First of all, huge thanks for buying **Anti-Cheat Toolkit**, 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 / or skilled cheaters are able to break anything!

IMPORTANT:
You can grab latest ACT version at any time – just drop me your Invoice ID and I'll send you latest version back (new version usually needs some time to appear at Asset Store). You'll know about new version from my twitter or at forums.

Anti-Cheat Toolkit has a full API documentation here:
http://codestage.ru/unity/anti-cheat/api/
Please, consider visiting it first if you have any questions on plugin usage.

Memory cheating.

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 other tools for these and other platforms as well.

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

```
ObscuredFloat
ObscuredInt
ObscuredString
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 ;)
For such cheating detection use ObscuredCheatingDetector, for example:

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

```
// set callback for cheating detection
// you may place it at Awake() or Start() handlers of MonoBehaviour for example
ObscuredCheatingDetector.StartDetection(OnObscuredTypeCheatingDetected);
// ...

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

Cheating detection currently implemented all obscured types, except ObscuredPrefs – it has own detection mechanisms.

Cheating detection will not work if you'll not call **StartDetection()** method, thus, fake variables will not exist in memory. 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 see examples of obscured types usage and try to cheat them yourself in the "TestScene", shipped with plugin. Feel free to write me if you wish to see any Obscured version of built-in type I did not implemented yet.

IMPORTANT:

- Currently only these types can be exposed to the inspector: ObscuredString, ObscuredBool, ObscuredInt and ObscuredFloat. Last three require Unity 4.5 or higher. Be careful while replacing regular types with the obscured ones – inspector values will reset! Obscured values inspector exposition is currently in beta state!
- All simple Obscured types have public static methods **GetEnctypted()** and **SetEncrypted()** to let you pull encrypted value from obscured instance. May be helpful in case you are using 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. or just keep an eye on your Profiler metrics, especially while working on mobile projects (I tried to use simple obscured variables in Update() on iPad1 without any noticeable performance drop though).
- LINQ is not supported at this moment.
- XmlSerializer is not supported at this moment.
- Some types are not compatible with Flash Player, make sure to check API docs of types you're going to use first.
- 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.

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 good old 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: Flash Player is not supported!

- 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 this callback could run 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 fired**.**

- OP supports some additional data types comparing to regular PP: double, long, bool, byte[], Vector3.

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 if you have server-side authorization, to lock saves to the unique user ID. Most actual on 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).

Once again, please keep in mind ObscuredPrefs will work slower comparing to the regular PlayerPrefs since it encrypts and decrypts all your data which costs some 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

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 the main scene using menu item "GameObject->Create Other->Code Stage->Anti-Cheat Toolkit->Speed Hack Detector".
You will see new Game Object in your scene named "Speed Hack Detector" with component attached to it:

This game object is able to nest in any other game objects,
it's not allowed to nest anything inside or put any other components there though.

Let me describe fields you see in inspector (**StartDetection()** method has same arguments):

Auto Dispose: Speed Hack Detector will self-destroy after speed hack detection.

Keep Alive: Speed Hack Detector's game object will remain in your app on new level (scene) load.

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 use 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)!

Here are few code examples:

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

// place it somewhere at the Awake() or Start()
// to call it once on Application startup
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:

```
// place it somewhere at the Awake() or Start() to call it
// once on Application startup
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" shots before cool down.

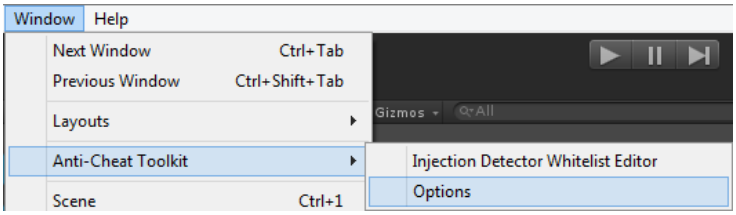
See the example of Speed Hack Detector set up in the TestScene and feel free to use prefab with it from "CodeStage\AntiCheatToolkit\Examples\Prefabs\Anti-Cheat Toolkit Detectors".

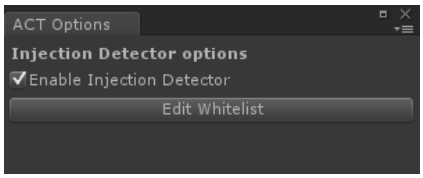
DLL Injection

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"





Set up is similar to the Speed Hack Detector: generally all you need to do - just call InjectionDetector.StartDetection() once anywhere in your code. Let me show you an example:

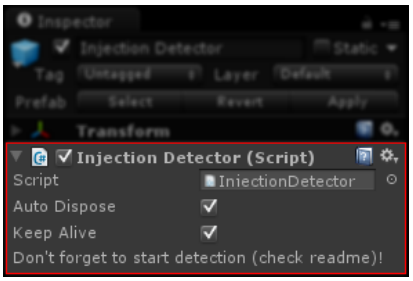
```
// place this line right in the beginning of your .cs file!
using CodeStage.AntiCheat.Detectors;

// this line makes few important steps under the hood:
// - creates Injection Detector game object in scene if it doesn't exists
// - starts detection engine
InjectionDetector.StartDetection(OnInjectionDetected);

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

Simple way to detect advanced cheat!

You also may set up detector in editor and set few additional parameters in inspector. In such case, you should add it to the main scene using menu item "GameObject->Create Other->Code Stage->Anti-Cheat Toolkit->Injection Detector".
You will see new GameObject in your scene named "Injection Detector" with component attached to it:



This game object is able to nest in any other game objects,
it's not allowed to nest anything inside or put any other components there though.

Settings you see in inspector are equal to same settings from Speed Hack Detector:
Auto Dispose: Injection Detector will self-destroy after detection reporting.
Keep Alive: Injection Detector's game object will remain in your app on new level (scene) load.
Don't forget you still need to call InjectionDetector.StartDetection() from code to start detector!

See the example of Injection Detector set up in the TestScene and feel free to use prefab with it from "CodeStage\AntiCheatToolkit\Examples\Prefabs\Anti-Cheat Toolkit Detectors".
Let me tell you few words on one 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 HOTween.dll + 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 may be covered manually (read below).

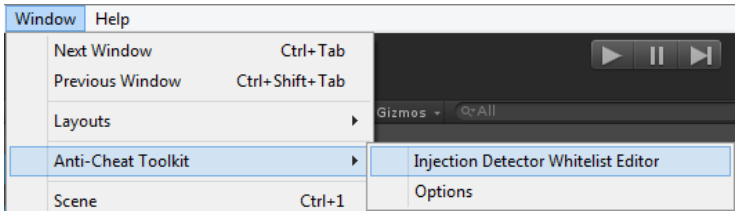
Detector leveraging whitelist approach, used to skip allowed assemblies. It generates such list automatically while you work on your project in the Unity Editor and stores it in the Resources/fn.bytes file.
This whitelist consists of these three parts:

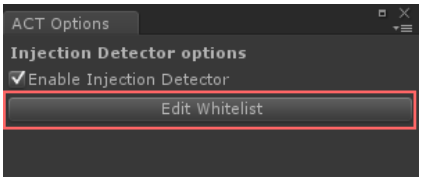
- default static whitelist (first group), stored in ServiceData/InjectionDetectorData/DefaultWhitelist.bytes file
- dynamic whitelist from assemblies used in project (second group), it updates automatically after scripts compilation
- user-defined whitelist (third group, see details below)

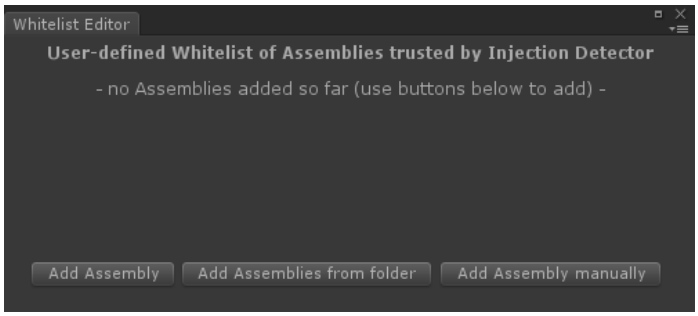
In most cases, you shouldn't do anything but just enabling Injection Detector in options and starting it in script to let it work properly.
However, if your project loads some assemblies from external sources (and these assemblies are not present in your project assets) or generates assemblies at runtime, 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:







- 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, lust enable debug in the Injection Detector (comment out #undef DEBUG line right in the beginning of the InjectionDetector.cs file) and let our assembly be caught by detector. You'll see its full name in console log, after "[ACT] 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 ServiceData/InjectionDetectorData/UserWhitelist.bytes file.

Few boring words

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 mostly tested on these platforms: **PC** (Win, Mac, WebPlayer), **Flash Player**, **iOS**, **Android**. I know some customers use it in projects for **Windows Phone**, so it should be fine there too.
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 a bit):
http://forum.unity3d.com/threads/anti-cheat-toolkit-released.196578/page-3#post-1573781

Plugin may be integrated with **PlayMaker** by hands. Example by kreischweide:
http://codestage.ru/unity/anti-cheat/playmaker_example.zip
Probably I'll make a kind of PlayMaker support package for Anti-Cheat Toolkit in future.

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 (links below)!

ACT links:

Asset Store | Web Site | Forum

Support contacts:

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

Best wishes,
Dmitriy Yukhanov
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** (HOTween, HOTools, Goscurry and many other happiness generating things creator) for awesome logos, intensive help and priceless feedback on this toolkit!*
P.S. #2 Below you may find outdated tutorial about cheating and using ACT against it. I'll wipe out or update it in future.

Veery outdated, feel free to ignore :o]

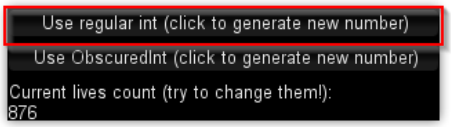
Testing cheatproof

This tutorial was created using pretty outdated version of plugin and demonstrates only few features from current version, but it still actual and things work similar to how it shown here.

Now let me show you a few screenshots of how cheaters can add a few lives or money for example (feel free to ignore this whole section, though I'd recommend you at least read it for your knowledge).
Feel free to watch a video version of this tutorial!

Memory cheating.

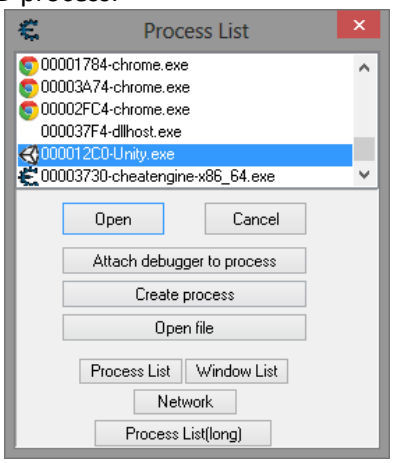
Most known tool for memory cheating is Cheat Engine. Download it!
Open TestScene, Run it, and click "Use regular Int":



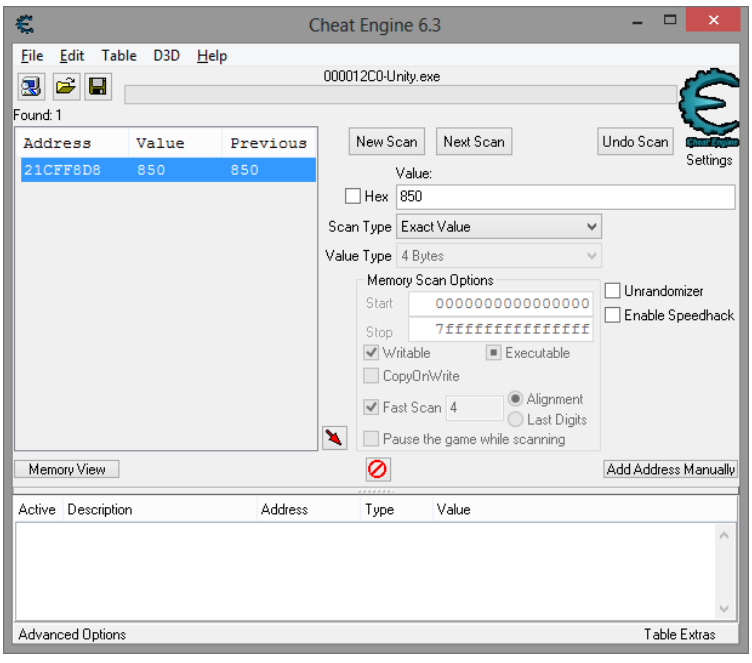
You can see new number generated below, 823 in this case. Now imagine this is gold (or whatever else, like lives count) in your game and someone (us in this case) wants to cheat it!

Here is a my version of cheating for dummies =)

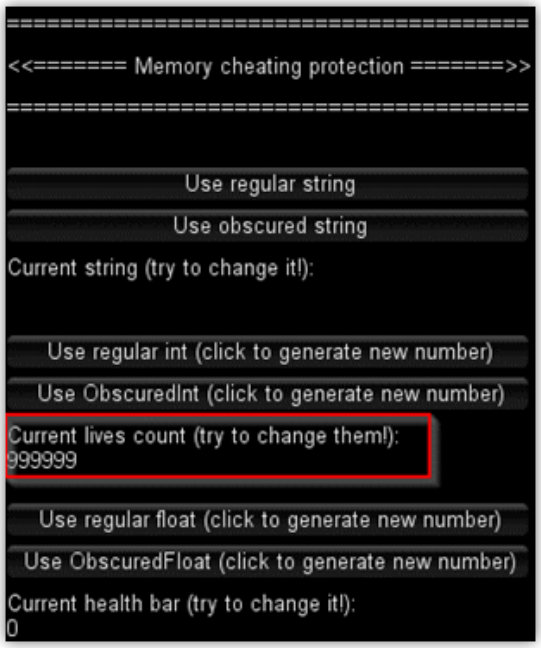
1. Open CheatEngine and **Open** Unity3D process.



2. Enter 823 in Value, make sure **Scan Type: Exact Value** and **Value type: 4 Bytes** are selected. Press New Scan, First Scan.
3. Click "Use regular Int" again (to change value, imagine the player just spent or earned some gold).
4. Enter new value in Cheat Engine.
5. Press Next Scan.
You'll see there are less addresses on the left side.
6. Repeat steps 4-5 until only 1 address remains on the left side.
I got it from second try (850).



7. Now just Right-Click that address, select "Change value of selected address", change it to something new, like 999999 (I want tons of gold!!!) and switch back to Unity:



This is some dummy sample cheating I just made, just to show you how easy it is.

Now, try repeating all these steps using Obscured type - click "Use ObscuredInt" instead of "Use regular int".
I bet you'll get stuck at step 6 :P

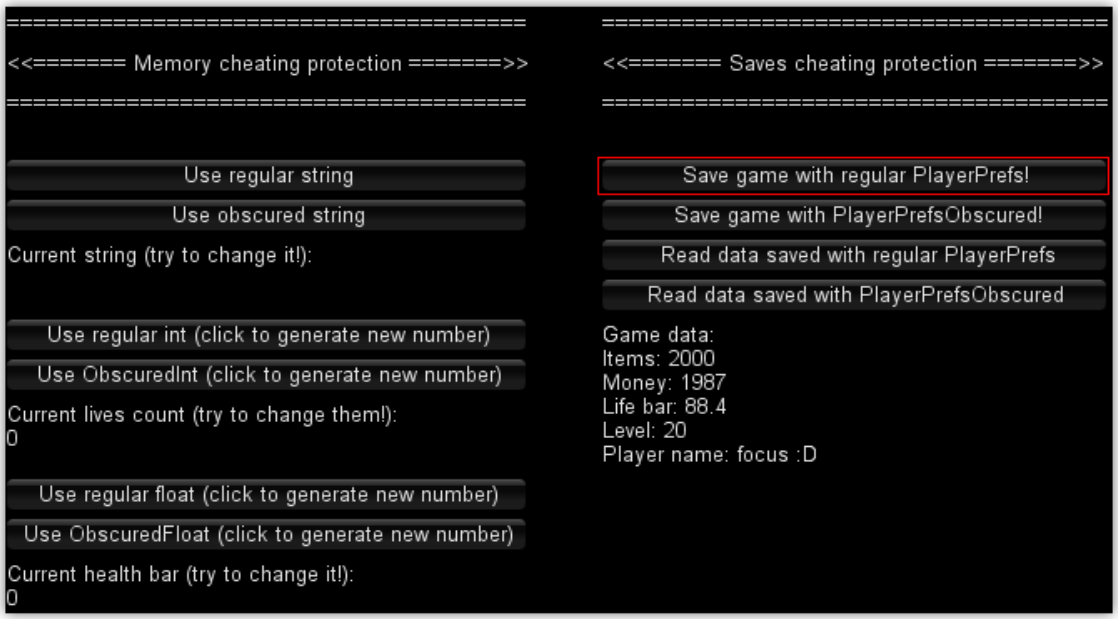
Saves cheating

I'd recommend to check out official Unity docs for the PlayerPrefs class first:
http://docs.unity3d.com/Documentation/ScriptReference/PlayerPrefs.html

Note: this page describes where PlayerPrefs stores saved data.
Since I'm a Windows user, I'll guide you next using Windows PlayerPrefs storage version.
So, data should be saved into the Windows registry, here in our case:

```
HKEY_CURRENT_USER\Software\Code Stage\Anti-Cheat Toolkit
```

Let's see what can we cheat there.
First, we need to save something, use "Save game with regular PlayerPrefs" for that:



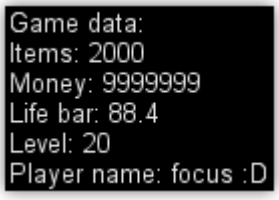
This button actually calls few PlayerPrefs.Set* methods under the hood, saving data usual way:

```
PlayerPrefs.SetInt("itemsFound", 2000);
PlayerPrefs.SetInt("currentMoney", 1987);
PlayerPrefs.SetFloat("currentLifeBar", 88.4f);
PlayerPrefs.SetInt("currentLevel", 20);
PlayerPrefs.SetString("currentPlayerName", "focus :D");
```

Here is how this data stored in Windows registry:

Name	Type	Data
(Default)	REG_SZ	(value not set)
currentLevel_h2066135418	REG_DWORD	0x00000014(20)
currentLifeBar_h3900298683	REG_DWORD	(bad value)
currentMoney_h2074476092	REG_DWORD	0x000007c3(1987)
currentPlayerName_h3100262872	REG_SZ	focus :D
itemsFound_h3587801141	REG_DWORD	0x000007d0(2000)

As you can see it can be easily accessed and edited by any person. Try to change something (currentMoney for example) and press the "Read data saved with regular PlayerPrefs" button to see changes are loaded:



I'm Richie Rich now! :D
Now try to press "Save game with PlayerPrefsObscured" and look at the saved data:

Name	Type	Data
(Default)	REG_SZ	(value not set)
T)"%└O87D_h2549184856	REG_DWORD	0x0000054a(1354)
*U>=3+T	REG_DWORD	0x00000559(1369)
*U>=3+T <V"%_h3232797817	REG_DWORD	0x0000028e(654)
*U>=3+T 0F"ß3>_h2753583398	REG_SZ	4635921014029550234
*U>=3+T5A>, (M)_h1443326272	REG_SZ	/O/:%e→

Well, you can see there is no chance to edit this data without breaking something at all!
Use "Read data saved with regular PlayerPrefsObscured" button to read this data back to see it was loaded correctly.