不正デバッグから.NET アプリケーションを防御する

AG-TECH CORP.®

目次

I.　はじめに 2

(1)公式マニュアルについて 2

II.　難読化だけでは動的解析が防げない 2

（１）　デバッグ　チェック 2
（２）　ルート　チェック 3
（３）　Shelf Life　チェック 3
（４）　改ざんチェック 3

III.　デバッグチェックを使用する方法 4

（１）　チェックを構成 4
（２）　チェックの動作を決めるさまざまなプロパティを指定 5
（３）　チェックの検証 7

IV.　まとめ 9

I. はじめに

Dotfuscator では、リバース エンジニアリングを困難にさせる手法だけではなく、コードの差し込みによってアプリケーションにチェック機能を追加できます。 アプリケーション開発者がコードを記述する必要はありません。Dotfuscator は適切な検証ロジックを自動的に差し込み、チェックが不正な使用を検出したタイミングに事前に作成したレスポンスを差し込むこともできます。このドキュメントでは、チェック機能の一つである「デバッグ検知」の機能をご説明いたします。

(1) 公式マニュアルについて

Dotfuscator の公式マニュアルでは、Dotfuscator を使用する上での詳細が記述されています。Dotfuscator を使用する上で、より詳しい内容は、WEB ページをご覧ください。

https://www.agtech.co.jp/download/manual/preemptive/

II. 難読化だけでは動的解析が防げない

アプリケーションをより強固に保護したい場合、リバースエンジニアリング対策の難読化だけでは不十分だと言えます。理由としては、攻撃者は起動後にプロセスにアタッチすることも考えられ、その場合、難読化では動的な攻撃は防ぐ事ができないためです。

もちろん、難読化されたアプリケーションをデバッグすることは難しくなりますが、ランタイム検出と防御は、静的な保護と組み合わせて、さらに効果的な抑止力を作ります。

Dotfuscator が提供している動的な検知機能として、下記が用意されております。

（1）デバッグ チェック

デバッガーで現在デバッグされている状態を検出して対応することができます。攻撃者が機密データを抽出または操作したりするために、デバッガー環境でアプリケーションを起動した場合には、デバッガーを検出し、攻撃者を妨害したりして対応することができます。

（2）ルート チェック

Xamarin.Android アプリケーションに限定されますが、ルート化された Android デバイスで実行されているコードを検出して対応することができます。攻撃者は、ルート化されたデバイスでアプリケーションを実行して、アプリケーションのバイナリへのアクセス、アプリケーションのリバース エンジニアリング、機密データの抽出、またはアプリケーションの動作の操作を行う可能性がありますが、デバイスがルートされていることを検出し、攻撃者を妨害したりして対応することができます。

（3）Shelf Life チェック

アプリケーションの特定のインスタンスの有効期限が切れていないかどうかを検証します。期限切れは、動的な開始日からの日数で指定することができます。開始日には、インストールの日付、アプリケーションが最初に実行された日付などが挙げられます。

（4）改ざんチェック

実行中のコードが、ビルドされた以降に改ざんされていないかどうかを検証します。攻撃者が規制を回避したりライセンス情報を削除したりするようにアプリケーションのバイナリを変更した場合には、改ざんチェックはそのような変更を検出し、攻撃者を妨害したりして対応することができます。

III. デバッグチェックを使用する方法

本資料では、デバッグチェックを導入する方法を紹介します。

全体の流れとしては、

・チェックを構成

・チェックの動作を決めるさまざまなプロパティを指定

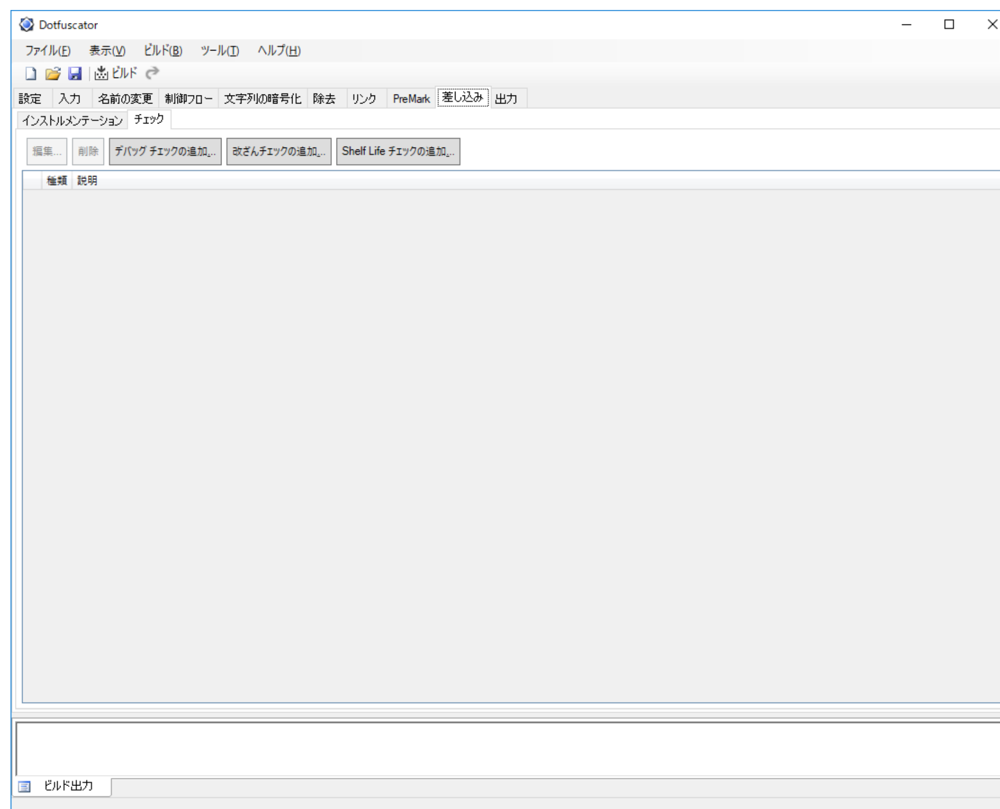
・チェックの検証

となります。

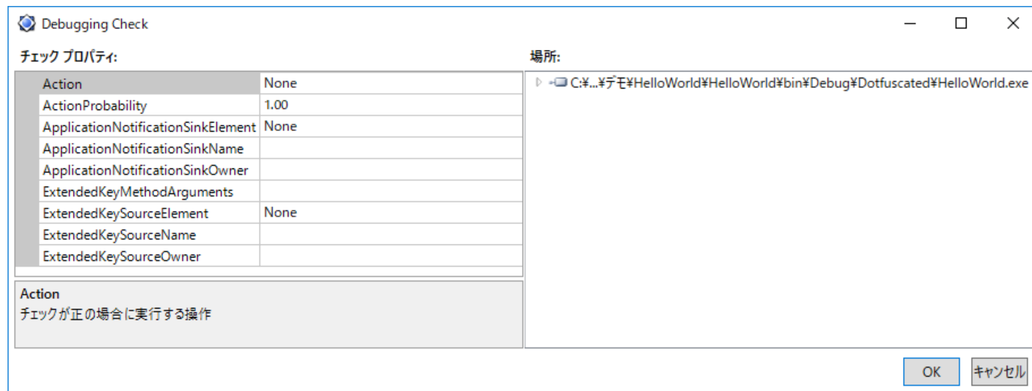
※ソースプログラムは書き換えませんので、ご安心ください。

（1）チェックを構成

Dotfuscator ウィザードのメニューより、[差し込み]タブをクリックします。

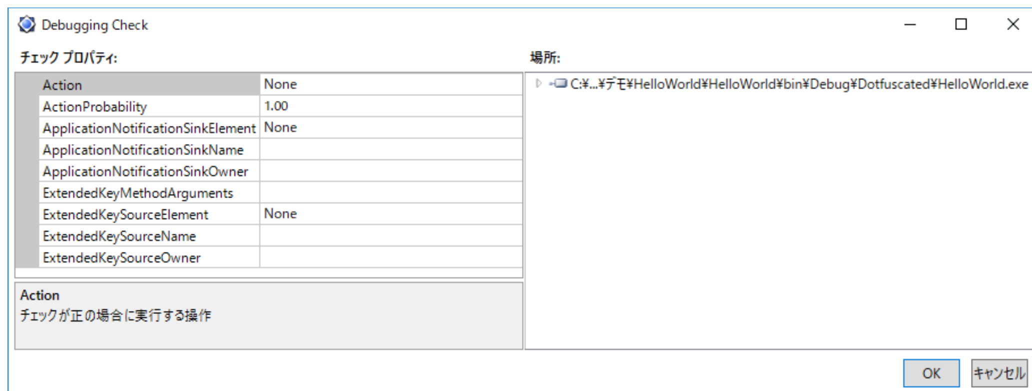


デバッグ チェックの追加ボタンをクリックします。



（2）チェックの動作を決めるさまざまなプロパティを指定

チェックの動作を決めるさまざまなプロパティを指定できます。



主なプロパティは下記になります。

Action：チェックが改ざんされたアプリケーションを検出したときに実行するチェック操作です。

ActionProbability：改ざんが検出されたときにチェック操作が行われる確率（0.00 ～ 1.00）を指定します。

※既定値は 1.00（デバッガがアタッチされている場合はアプリケーションを 100%終了させます）

今回は Form1 のボタンをクリックした場合、毎回アプリケーションを異常終了させるように設定します。

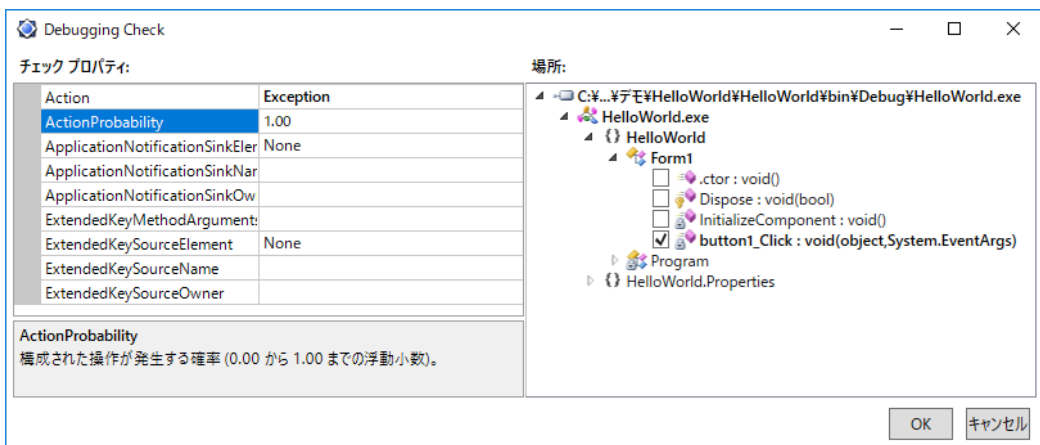
場所:

button1_Click にチェック

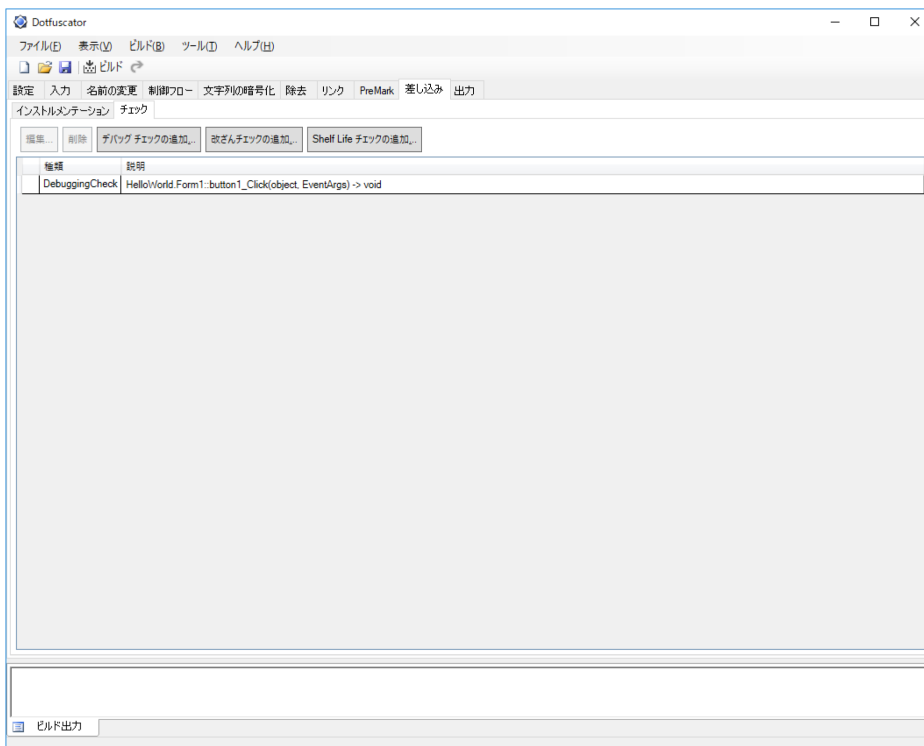
チェック プロパティ:

Action : Exception

ActionProbability:1.00



DebuggingCheck が追加されました。



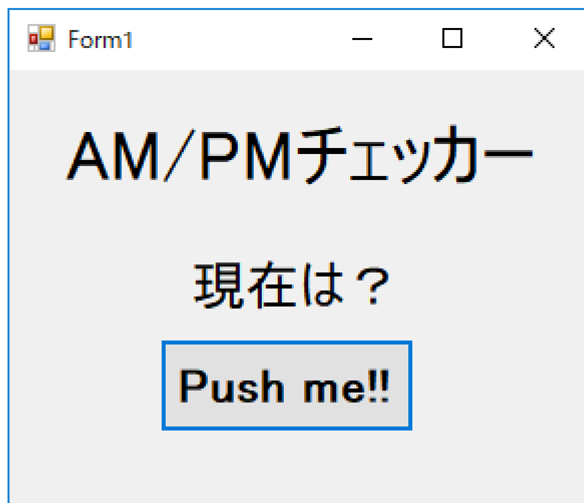
アプリケーションをビルドします。

ビルド出力画面にてデバッグチェックが差し込まれたことがわかります。

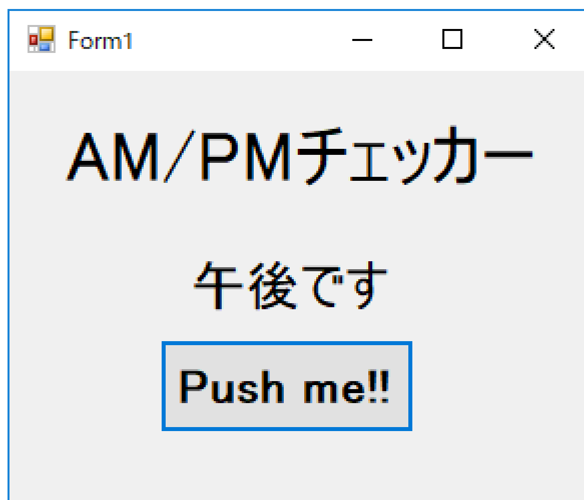
DebuggingCheck: HelloWorld.Form1::button1_Click

（3）チェックの検証

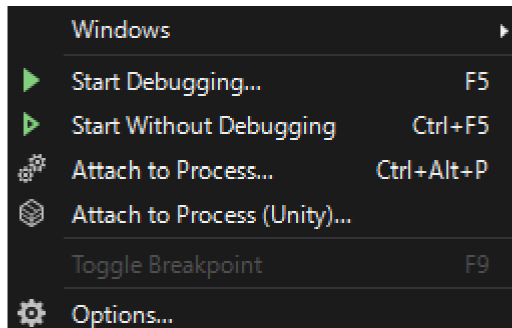
Dotfuscator によってビルドされたアプリケーションを立ち上げます。

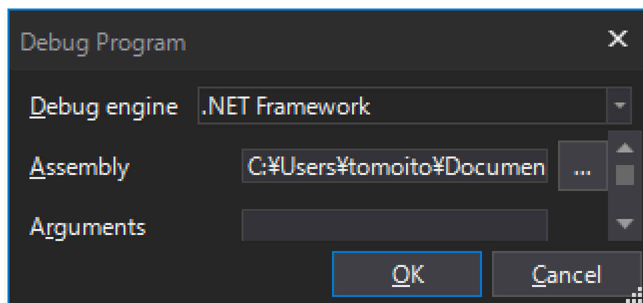


ボタン押下すると、ラベルの表示が変わります。

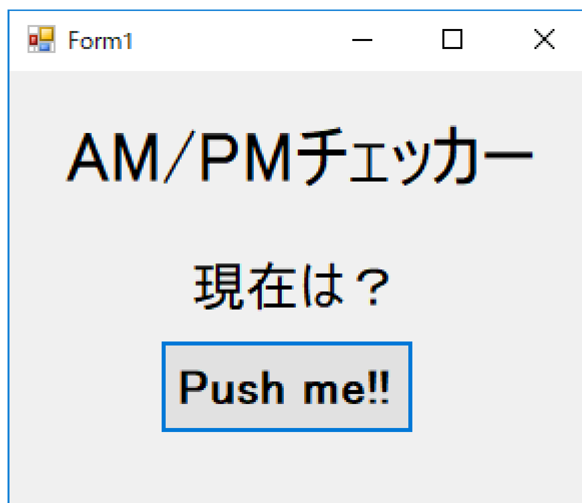


あるデバッグソフトを利用して、アプリケーションのデバッグをします。

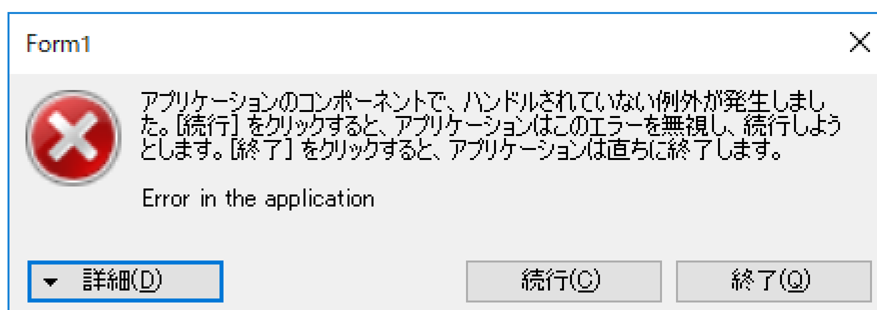




アプリケーション起動後は、何も変更がありません。



ボタン押下後、下記の異常終了メッセージが表示されました。



このように、難読化では対処ができない実行時の解析から、アプリケーションを保護することができます。

IV. まとめ

難読化が、「静的解析」（つまり、アセンブリ ファイルの解析）からアプリケーションを保護するのに対し、チェックは、実行されている間の「動的解析」からアプリケーションを保護します。二つの機能を組み合わせることで、攻撃者が目標を達成することをますます難しくさせます。今回はデバッグチェックを紹介しましたが、その他のチェック機能も簡単に導入ができ、強力なプロテクションを提供します。より強固にアプリケーションを保護したい方は、弊社が販売している Dotfuscator または DashO をご検討ください。

※Dotfuscator ⇒ NET 開発環境向け難読化ツール

DashO ⇒ Java/Android 開発向け難読化ツール