SecHack365

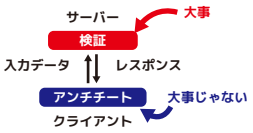
ゲームのチート対策を本質的にする社会実験手法の提案

開発駆動コース　仲山ゼミ　平田麟太朗

あなたはチート対策の本質が何か、答えられますか？

様々なゲームが登場し、チートやその対策が行われる昨今ですが、その対策手法はどれも新たなバイナリやプロセス、ドライバを監視し、シグネチャとの一致を検出するなど、ウイルス対策と同じ手法が使われています。しかし、これはウェブにおける公開鍵暗号化方式を用いた通信とは違い、理論的な堅牢性が担保されていません。しかし、不正を行えない環境としてチート対策を見ると、その本質は「明確にクライアントとサーバーを分離すること」だけで、決して「アンチチートを導入すること」ではないと気づくはずです。

本質的な不正対策



既存のアンチチートの問題

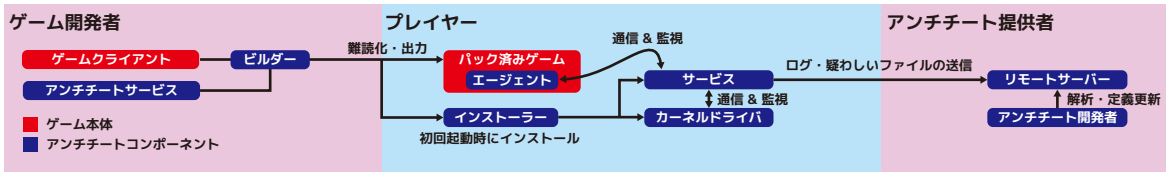
既存のアンチチート（EAC、BattleEye、Vanguard）は、カーネルモードとして動作するものが多く、アンチチートと言う大義がなければ、実質的なルートキットです。そのユーザー数の多さから、脆弱性発見時の影響度は高い上に、任意のファイルをアンチチートのサーバーに送信できることから、プライバシーの侵害をしていると言う懸念も存在します。（怪しい挙動のファイルがのサーバーに送信されていることから）

ゲーム解析ツールが社会に与えうる影響とは

ブラックボックスが本当にあなたのゲームを守ってくれると思いますか？

これらのアンチチートはオープンソースではない上に、難読化が施されてゲームバイナリに埋め込まれています。この場合ブラックボックスであることがゲームを守る手法として使用されているわけです。不正者によるリバースエンジニアリングを防ぎ、不正行為の気力を削ぐ目的ですが、理論的堅牢性を備えているわけではないため、解析されてしまえば全くの無防備になってしまいます。Apex Legends のようなチートを作ることで大金を稼ぐことのできるゲームであれば尚更解析の心理的ハードルは下がります。仮に社会にそのようなブラックボックス化を無効化し、改造を自由に行えるようにするツールが登場したらどうでしょうか？ゲーム開発者は不正者からゲームを守るために、上記で書いたチート対策の本質である、明確にクライアントとサーバーを分離するようになるはずです。

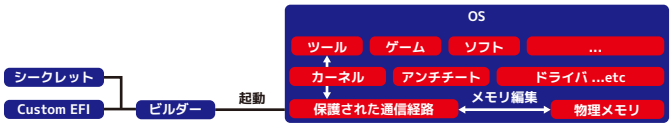
一般的なアンチチートの構成



誰もがアンチチートで保護されたゲームを解析することができるツールであり、社会実験のメタファー。

アンチチートで保護されたゲームをリバースエンジニアリングするツールの手法を提案します。これは、任意のエンジンやライブラリを使用して作成されたゲームに対応できます。メモリの読み書きやシグネチャの自動解析を行うことができますが、アンチチートに検知されることはありません。既存の著名なツールや手法を解析し、様々なアプローチを試しました。OSS として有効である方法論について考察します。現状、ゲームを解析、改造するにはアンチチートと同様のレイヤー（Ring0、カーネル空間）でプログラムを動かす手法が主流です。この手法の場合、アンチチートのフックを更に再フックすることや、プロセスの保護を無視できるため、より容易に解析やアンチチートのバイパスが可能になりました。しかし、これは公開ツールという観点では、アンチチート側が OS にロードされているドライバを監視するだけで検知が出来るため、好ましくありません。ドライバに存在する脆弱性を利用し、任意コードをカーネル空間から実行することも出来ますが、公開してしまうとその脆弱性をアンチチート開発者も解析し、対策を打つことができ、堅牢な解析ツールとは言えません。そのため、EFI 領域で動作する物理メモリ読み書き用レイヤーを用意するという、OSS としては新たな手法を提案します。アンチチートは、その性質上チートを防ぐという名目でも EFI 領域にインストールさせることはユーザーの理解を得ることが出来ないため、非常に困難です。しかし、ユーザーが解析をしたいと言う意図を持ってオープンソースのツールを自ら EFI 領域に導入する場合、アンチチートの場合と比較して参入障壁は低いと言えます。ユーザーランド側のツールと通信を行う場合、何も保護していなければアンチチートがツールなりすまし、EFI が改造されているか確認できてしまうため、EFI 領域で動作する実行ファイルをビルドするツールを内包し、ユーザーが各々シークレットを設定した上で EFI をビルドし、ユーザーランドのアプリケーションが同じシークレットを保持しないと動作しないようにすることで、暗号学的堅牢性を担保できます。以上が OSS として有効であると言えるゲームの動的解析ツールの説明です。

ツールのアーキテクチャ



既存手法 1

第一の手法として、外部からシステムコールを使用してメモリを読み書きする方法が挙げられます。Windows であれば、WindowsSDK の {Read, Write}ProcessMemory 関数を使用し、プロセスのハンドルを取得することでメモリの読み書きが可能です。しかしアンチチートは、その特権を使用して Windows の API をフックすることや、ハンドルを常に監視することでその動作を検知することが出来ます。

既存手法 2

第二の手法として、ゲームプロセスに DLL を注入する方法が挙げられます。様々な方法がありますが、どれもプロセスの仮想メモリに新たなモジュールを用意し、スレッドを開始するとで内部の値を読み書きします。アンチチートは常にプロセスにロードされているモジュールを監視しているため、不審なモジュールは常にフラグが立てられます。

既存手法 3

第三の手法として、フックがあります。基本的には第二手法で DLL を注入した後に使用しますが、任意の関数ののアドレスやリターンアドレスを書き換えることで、処理前や後に好きなコードを実行できるようにするものです。アンチチートは、不正なアドレスが何かを知っているため、検知されます。

メリット

これらの技術は、低レイヤーやアンチチートに精通していなくても使用できるといったメリットがあります。あくまでアーキテクチャなので、既存のツールと組み合わせることで特定のゲームにおいても便利に改造等を行えるようになります。OSS なので、その挙動を自分で調べるといったことはもちろん、プラグインのような仕組みで拡張をしたり、脆弱性の修正等の貢献をすることが可能です。

本質的な不正対策が生む、ゲームのあるべき姿

本質的な不正対策とは、ウェブサービスと同じです。クライアントからの出力は改ざんされる前提で、暗号学に基づいた信頼境界を定め、サーバーで処理を行う。チート対策のために不必要な特権をプレイヤーのコンピュータでは知らせることは不要で、アーキテクチャを正しくすべきなのです。最も良い例はマインクラフトでしょう。Mod が当たり前の本作ではクライアントがどのような改造を行っているかは重要ではなく、プロトコルとして正しさを判定し、サーバーはそれを処理する。サーバー側で不正な行為やパケットを検知すると言ったアンチチートがゲームのあるべき姿だと考えました。

NICT