GHB4 - Anti-Debug, AntiCheat & Kernel Mode

This is not a complete part of the Game Hacking Bible. In it's place, here is a collection of our resources related to these topics.

Do not be naive and think bypassing anticheat is easy.

Main Resources

- 300+ Threads in the Anti-Debug & Anticheat Forum Section
- Guide - How to Bypass Anticheat - Start Here Beginner's Guide
- Tutorial - Junk Code Generator and Polymorphic Code Engine Guide
- Guide - How To Bypass VAC Valve Anti Cheat Info
- Guide - Kernel Mode Drivers Info for Anticheat Bypass
- Guide - Anticheat Battleye Bypass Overview
- Guide - How to bypass XignCode Anticheat Guide - XignCode3
- Guide - Hackshield Anticheat Bypass Information
- Guide - How to Bypass FairFight Anticheat
- Download - GamersClub Anti-Cheat Information (Driver + user mode module)
- Tutorial - MTA: SA's kernel mode anticheat is a joke (information)
- Guide - Anticheat XTrap Bypass Source Codes
- Guide - Anticheat nProtect Gameguard Bypass

Additional Resources

- Tutorial - Bypassing anti debug example in CS2D
- Solved - How to Bypass Ragnarok Anticheat - Gepard Shield Bypass
- Source Code - CVEAC-2020: Bypassing EasyAntiCheat integrity checks
- Download - Easy Anti-Cheat - EAC Driver Dumps - Unpacked Modules

Offsite Resources

- secret club
- Reversing XignCode3 Driver - Part 4.2 - Verifying windows version - Niemand - Cyber Security

How to Bypass Anticheat - Start Here Beginner's Guide

Anti-Cheat Bypassing Guide for Noobs

Introduction

This thread contains a rough overview of information and skills you will need to bypass anticheat. There is also a number of important links and references that you will need as you learn more about anticheat. You should not even think about attempting to bypass anticheat until you have at least 6 months experience.

Do not learn game hacking on games with anticheat, this is a waste of time.

Instead, learn game hacking first on easy games. Then when you're adequately experienced, start learning about anti cheat using this guide and then work on reversing and bypassing an anticheat.

Kicked or Crashed When Attaching Cheat Engine?

This means they are detecting the Cheat Engine string or the debugger attaching. These are the first things to try if the game doesn't have a commercial anticheat.

The first and easiest steps to attempt to bypass anticheat are:

- Use VEH Debugger in Cheat Engine (it's in options under Debugger)
- Try Undetected Cheat Engine
- Try Cheat Engine Alternatives
- Inject Scylla Hide first (or use x64dbg plugin)
- Try using Manual Mapping and other injection methods from the GH Injector

What is Anticheat?

Anticheat is functionality built into the game or additional software that runs while the game is running, it uses various methods to detect cheats. You typically cannot play the game without it running. Most of the functionality built into anticheat is just classic antidebug with signature detection of cheats that the anticheat has built signatures for.

Features Anticheat Uses

- File Integrity Checks

- String Detection for cheat tools
- Classic AntiDebug
- Obfuscation
- Signature Based Detection
- Hook Detection
- Memory Integrity Checks
- Virtualization
- Kernel Drivers which block process access token creation & more
- Virtualization Detection

Home Rolled Anticheat

Game developers can easily implement the first few anticheat features, specifically file integrity checks, string detection for cheat tools & classic antidebug. These are relatively easy to bypass.

Commercial Anticheat

If you're a developer you can purchase/subscribe to thirdparty commercial anticheat. These will always be more strict and more difficult to bypass than any anti-debug that the developer creates themselves. The most common commercial anticheats are Battleye, EasyAntiCheat & Xigncode.

Other common anticheats include Punkbuster, Fairfight & Hackshield.

Valve Anti Cheat

This is the worst anticheat on the market, do not worry about stupid VAC unless you're selling paycheats. Everyone asks stupid question about VAC as if it was some god tier anticheat, it's trash and is bypassed without doing anything special. Read more @ Guide - How To Bypass VAC Valve Anti Cheat Info

The most important thing you can do to understand anticheat is watch this playlist:

https://www.youtube.com/watch?list=PLt9cUwGw6CYG-d7LGlLKHmLWFBJqA2XSV&v=yJHyHU5UjTg

Anticheats have the capability to detect every single thing that occurs on your computer, they are extremely invasive, all kernel anticheats are essentially rootkits. Even VAC scans every single process that's running. The question is, do they have a signature or other detection vector for your specific cheat. Signatures are built for known cheat software, so if you write your own software, they can't detect it based on signature. They can still use heuristics, but they won't autoban for heuristics unless it's very obvious it's a cheat. If you're not distributing your hack to 15+ people they are not gonna waste their time analyzing your cheat in most cases. They have limited resources like every business

GH Specific Anticheat Guides

The are all our guides related to this thread, check these out after you read this guide

- Guide - How to Bypass EAC - Easy Anti Cheat
- Guide - Anticheat Battleye Bypass Overview
- Guide - How to Find Encrypted or Obfuscated Variables in Cheat Engine Guide
- Tutorial - Junk Code Generator and Polymorphic Code Engine Guide
- Guide - Kernel Mode Drivers Info for Anticheat Bypass
- Guide - How To Bypass VAC Valve Anti Cheat Info
- Guide - Battleye Anticheat Bypass Overview
- Guide - How to bypass XignCode Anticheat Guide
- Guide - Hackshield Anticheat Bypass Information

- Guide - How to Bypass FairFight Anticheat
- Download - GamersClub Anti-Cheat Information (Driver + user mode module)
- Tutorial - MTA: SA's kernel mode anticheat is a joke (information)
- Guide - CSGO Overwatch Bypass - How to Avoid Overwatch Bans

How to bypass anticheat?

There is no magic trick or download we can give you to instantly bypass anticheat. If you have been game hacking for less than 6 months, you have no business asking about anticheat. You cannot even understand because you do not have the required knowledge to do so. If we told you how to bypass anticheat you wouldn't be able to implement it because it's not a step 1-2-3 process.

If you want to bypass an anticheat from scratch, by yourself you need 6-24 months experience game hacking. If you want to bypass anticheat by pasting, gtfo.

To bypass anticheat you must hide from it, disable it, bypass it or spoof the results of it's checks. Anticheats will use multiple methods to detect you and multiple methods to protect itself, so it's not typically as easy as bypassing one feature and you're done. It's usually a multi-pronged approach.

The second more difficult steps to attempt to bypass an anticheat are:

- Don't use any public source code, write it yourself
- Do not share your hack with anyone
- Use manual mapping to inject using the GH Injector
- or better yet write your own manual mapping injector

How to learn to bypass anticheat

Here is a step by step guide on what your journey to bypassing anticheat should look like:

1. Guide - START HERE Beginners Guide to Learning Game Hacking
2. Guide - Beginners Guide To Reverse Engineering Tutorial
3. Practice & get experience hacking games without anticheat for at least 6 months
4. Make fully featured aimbots & ESPs for games without anticheat
5. Be moderately experienced with all aspects of object oriented programming
6. Read this guide you are currently reading
7. Learn anti-debug
8. Learn Important Aspects Windows Internals - you will know which parts are important
9. Guide - Kernel Mode Drivers Info for Anticheat Bypass
10. Reverse the anticheat of your choosing for several weeks
11. Create your anticheat bypass

If you skip more than 1 of these steps you will fail.

Steam AntiDebug / DRM

Publishers can opt in to have Steam add antidebug/DRM protection when releasing on Steam
Example of Steam antidebug spraying the stack when debugger is attached
ThreadHideFromDebugger bypass that all steam antidebug uses
More info here

Kernel Mode Anticheat Bypass

Read full thread: Guide - Kernel Mode Drivers Info for Anticheat Bypass

The Windows Operating System has different layers which we call rings, your game and your hacks are usermode ring 3 processes. Drivers such as your video card drivers run in kernel mode or ring 0. These drivers use a different API and are written using the Windows DDK (Driver Development Kit). These usually have the .sys file extension. They run BELOW usermode processes, usermode processes can't touch them. If the anticheat has a kernel mode driver you cannot patch it from usermode, you must either avoid detection or make your own kernel mode driver.

If you're 1337 you can use vulnerable drivers such as CapCom to load your system driver which you can then use to bypass kernel mode anticheats.

Here's a little guide: EvanMcBroom/EoPs

You can also defeat Protected Processes Light protection using Mattiwatti/PPLKiller which can sometimes enable you to be able to access and modify previously protected processes.

How Does AntiCheat Work?

To bypass anticheat you must understand how it works. Anticheat work very similarly to Antivirus. These are the basic things it does to stop you from cheating, kinda going from simple to more advanced

- File Integrity Checks
- Detecting Debuggers
- Stops debugger from attaching
- Detect Cheat Engine & memory editors
- Signature Based Detection
- Detect DLL injection
- Detect Hooks
- Block Read/WriteProcessMemory
- Memory integrity checks
- Statistical Anomaly Detection
- Heuristics

File Integrity Checks

Patching or hexediting the game.exe should never work. This is how custom minecraft clients work, you just make your own EXE or edit the one you get with the game. It is very simple to stop this, use a MD5 or SHA hash for all the important game files. If bytes in the .exe are changed the hash will change. when your game.exe loads it should compare the hashes of the important game files against a DB of file hashes, if hashes don't match, the game should close.

Bypass: To bypass File Integrity checks, only modify memory, not the files on disk. Or reverse engineer the integrity checks and patch them.

Most anti-cheats use signature based detection and file hashes. If a DLL gets injected with a known cheat file hash, you're cheating. Signatures are built for cheats in the same way that you build a pattern for a pattern scan or an antivirus detects viruses. To bypass signature and hash detection is too easy, write your own hacks and don't share them. Don't use public code that may match a signature that they already have. #1 most important is don't copy and paste, if you find some cool code you want to use, re-write it differently. I typically do this with everything because I learn it better and like my code to all have the same style.

Then you have detour/hooking detection. How to detour? you place a jmp in the instruction of a function, typically

hackers are hooking the same core functions such as directx endscene or lets say the gun::shoot() function. So they compare what's loaded into memory with what's written on disk, if the code doesn't match then it's obvious someone is modifying the code at runtime in memory. they could even just scan for jmps at 0x0 of every function. How about vtable hooks? just as easy to detect.

A decent way to make undetected ESP would be to make external, only use readprocessmemory and do an external overlay ESP, this would be undetected against most basic anticheats.

@timb3r's Anticheat Series

Tutorial - How to write an Anti-Cheat Part 1: Detecting External Cheats

Additional Resources:

- Developments | Cra0kalo's Development Adventures
- Anti-Debug Protection Techniques: Implementation and Neutralization
- An Anti-Reverse Engineering Guide
- Anti Reverse Engineering Reference PDF
- Solved - Anti-debug game
- Polymorphic code and Junk Code Generators

Games that use EAC (EasyAntiCheat)
7 Days To Die, Aboslver, Audition, Battaltion 1944, Block N Load, Cabal Online, Combat Arms, Crossout, Cuisine Royal, DarkFall: Rise of Agon, Darwin Project, Days of War, Dead by Daylight, Death Field, Dirty Bomb, Dragon Ball Fighter Z, Xenoverse 2, Dragonica Lavalon Awakens, Empyrion, Far Cry 5, Fear the Wolves, For Honor, Fortnite, Friday the 13th, Gigantic, Hide & Hold Out, Hunt Showdown, Hurtworld, Infestation: Surviror Stories, Infesntation: World, Intershelter, iRacing, Ironsight, Lifeless, Luna, Magicka Wizard Wars, Memories of Mars, Miscreated, Next Day, Offensive Combat Redux, Onward VR, Paladins, Post Scriptum, Ragnarok, Realm Royale, Reign of Kings, RF Online, Rising Storm 2, Robocraft / Royale, Rockshot, Rust, Sky Noon, Smite, Squad, Sword Art Online, Tales Runner, The Culling 1& 2, Ghost Recon Wildlands, Total War Arena, War of the Roses, War of the Vikings, War Rock, Warface, Warhammer 40,0000, Watch Dogs 2, World Adrift, Yulgang

Games that Use BattlEye
ARMA II, ARMA III, DAYZ, H1Z1, Ark Survival Evolved, Surivial Of the Fittest, PlanetSide 2, Rainbow Six Siege, Survarium, Project Argo, Unturned, Insurgency, Day of Infamy, The Isle, Line of Sight, Conan Exiles, Blacshot, Tibia, PUBG, Black Squad, Pantomers, Fortnite, S4League, Zula, Islands of Nyne, BlackLight Retribution, SOS, PIxark, Heroes & Generals, Bless Online.
Main Guide: Guide - Battleye Anticheat Bypass Overview

Nexon Game Security Bypass
Guide - Nexon Game Security Bypass info + hot sticky sauce

Hook Detection

anti debugging tricks follow jumps detect hooks

Continue reading the posts below for more information

Classic AntiDebug = Detecting Debuggers

All anticheats will probably use this technique. When you attach Cheat Engine or a debugger it uses a very specific

method of interacting with the target process. Windows operates this way for security. When you attach a debugger you're actually registering the debugger with the Windows OS, so detection is obviously quite easy.

These are 4 basic methods to detect a debugger

IsDebuggerPresent()

A simple Windows API function that will return TRUE if the calling processor has a debugger attached. They can just call this function and close the program if it returns TRUE. Read more here

This code will patch IsDebuggerPresent externally so it returns false every time

C++:

```
#include <iostream>
#include <Windows.h>
#include <TlHelp32.h>

DWORD GetProcId(const char* procName)
{
    DWORD procId = 0;
    HANDLE hSnap = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
    if (hSnap != INVALID_HANDLE_VALUE)
    {
        PROCESSENTRY32 procEntry;
        procEntry.dwSize = sizeof(procEntry);

        if (Process32First(hSnap, &procEntry))
        {
            do
            {
                if (!_stricmp(procEntry.szExeFile, procName))
                {
                    procId = procEntry.th32ProcessID;
                    break;
                }
            } while (Process32Next(hSnap, &procEntry));

        }
    }
    CloseHandle(hSnap);
    return procId;
}

uintptr_t GetModuleBaseAddress(DWORD procId, const char* modName)
{
    uintptr_t modBaseAddr = 0;
    HANDLE hSnap = CreateToolhelp32Snapshot(TH32CS_SNAPMODULE | TH32CS_SNAPMODULE32,
procId);
    if (hSnap != INVALID_HANDLE_VALUE)
    {
        MODULEENTRY32 modEntry;
        modEntry.dwSize = sizeof(modEntry);
        if (Module32First(hSnap, &modEntry))
        {
            do
            {
                if (!_stricmp(modEntry.szModule, modName))
                {
                    modBaseAddr = (uintptr_t)modEntry.modBaseAddr;
                    break;
```

```
                }
            } while (Module32Next(hSnap, &modEntry));
        }
    }
    CloseHandle(hSnap);
    return modBaseAddr;
}

void PatchEx(BYTE* dst, BYTE* src, unsigned int size, HANDLE hProcess)
{
    DWORD oldprotect;
    VirtualProtectEx(hProcess, dst, size, PAGE_EXECUTE_READWRITE, &oldprotect);
    WriteProcessMemory(hProcess, dst, src, size, nullptr);
    VirtualProtectEx(hProcess, dst, size, oldprotect, &oldprotect);
}

int main()
{
    DWORD procId = GetProcId("CS2D.exe");

    HANDLE hProc = OpenProcess(PROCESS_ALL_ACCESS, NULL, procId);

    //mov eax, 0
    //ret
    PatchEx((BYTE*)IsDebuggerPresent, (BYTE*)"\xB8\x0\x0\x0\x0\xC3", 6, hProc);

    std::getchar();
    return 0;
}
```

CheckRemoteDebuggerPresent()

Does the same thing but can work against an external process, so the game can run a separate process that calls this on the game process or it can just call it against itself. Read more here

Manually Checking the Being Debugged Flag in the PEB

Both of these functions read the BeingDebugged flag in the PEB (Process Environment Block). If you have bypassed the 2 above functions, they can manually read it from the PEB to bypass your hooks.

C++:

```
typedef struct _PEB
{
    BOOLEAN InheritedAddressSpace;
    BOOLEAN ReadImageFileExecOptions;
    BOOLEAN BeingDebugged; //<--
    //etc...
}
```

How to get PEB Address internally

C++:

```
__readfsdword(0x30); //x86
__readgsqword(0x60); //x64
```

You read the fs segment register offset 0x30/0x60 that gives you address of the PEB. Offset 0x3 of the PEB is the BeingDebugged flag.

How to get the PEB externally using NtQueryProcessInformation

C++:

```
typedef NTSTATUS(__stdcall* tNtQueryInformationProcess)
(
    HANDLE ProcessHandle,
    PROCESSINFOCLASS ProcessInformationClass,
    PVOID ProcessInformation,
    ULONG ProcessInformationLength,
    PULONG ReturnLength
    );

tNtQueryInformationProcess NtQueryInfoProc = nullptr;

bool ImportNTQueryInfo()
{
    NtQueryInfoProc = (tNtQueryInformationProcess)GetProcAddress(GetModu
leHandle(TEXT("ntdll.dll")), "NtQueryInformationProcess");
    if (NtQueryInfoProc == nullptr)
    {
        return false;
    }
    else return true;
}

PEB GetPEB()
{
    PROCESS_BASIC_INFORMATION pbi;
    PEB peb = { 0 };
    if (!NtQueryInfoProc) ImportNTQueryInfo();

    if (NtQueryInfoProc)
    {
        NTSTATUS status = NtQueryInfoProc(handle, ProcessBasicInformation, &pbi,
sizeof(pbi), 0);
        if (NT_SUCCESS(status))
        {
            ReadProcessMemory(handle, pbi.PebBaseAddress, &peb, sizeof(peb), 0);
        }
    }
    return peb;
}
```

How to Bypass these basic debugger detection techniques

All 3 of the above detections are based on the PEB.BeingDebugged flag, so you can bypass them all just by overwriting the BeingDebugged flag with 0.

You can also hook each function individually and change the return value to a spoofed result.

NtQueryInformationProcess with ProcessDebugPort argument

"Retrieves a DWORD_PTR value that is the port number of the debugger for the process. A nonzero value indicates that the process is being run under the control of a ring 3 debugger."

Bypass: Hook NtQueryInformationProcess in the game process

I have a little POC on the above methods
More info below from Roman_Ablo

ThreadHideFromDebugger

Tutorial - How to Find Hidden Threads - ThreadHideFromDebugger - AntiDebug Trick

Force an Exception and Try to Catch It

They can also force an exception to occur and try to catch it, if there is a debugger attached the exception will get caught by your debugger instead of the program.

Additional information on debuggers:

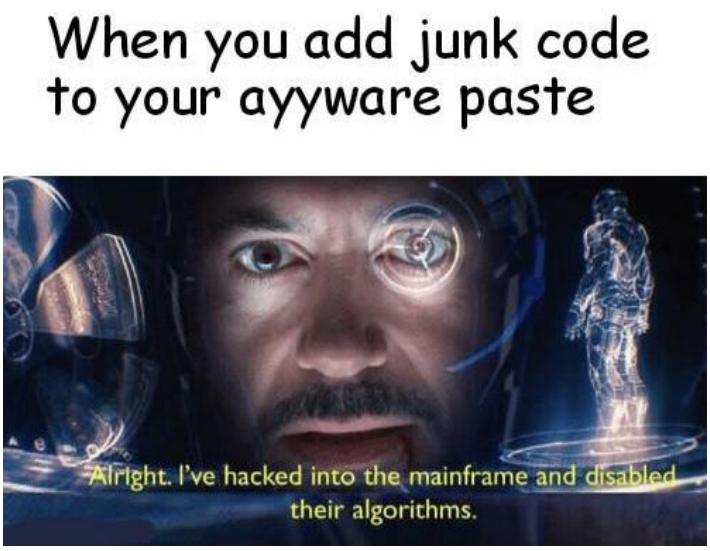
Basic Debugging (Windows)
How Windows Debuggers Work | Microsoft Press Store
Debugger Basics
Writing a basic Windows debugger - CodeProject

Junk Code Generator and Polymorphic Code Engine Guide

Junk code and Polymorphic code are both methods used to bypass hash based and signature based detection of your hacks by anticheats.

There are 2 important posts directly below this one from @mambda and @Liduen which contain actual source code for polymorpism

Junk Code Generator and Polymorphic Code Engine Guide



This is a guide to help you understand junk code / polymorphic engines. I do not know how to make one, but here is everything I know about the topic to get you started. At the bottom you will find from @mambda @c5 and @Liduen who have provided source code on this topic. If you're not an experienced coder/hacker, read the tutorial but don't expect that you'll be able to actually do this anytime soon.

Adding junk code will not bypass signature detection -> You must use polymorphic code

Prerequisites:

Please learn how pattern scanning / signature based detection works by doing this tutorial:

https://youtu.be/jLfPdujSuRA

Please learn about anticheat by reading our AntiCheat Guide
and our Valve Anticheat Guide for more general anticheat info

What is junk code?

Junk code is when you add code to your project that has no effect on the functionality of your project but will result in new assembly code in the binary after you compile.

Why would I want to add junk code?

Adding junk code to a project will change the file hash and hash of the code sections of the binary. Anticheats can make hashes of code sections to identify your hacks.

What is a hash?

Hashing algorithms take data and make unique hashes to identify this data, that is substantially smaller than the actual data size making identifying the data faster. For simple file hashes in the past CRC and mad5 have been used but SHA2 is the standard right now. To make a SHA256 hash is very simple you can just do

C++:

```
#include "sha256.h"
std::string input = "data";
std::string hash = sha256(input);
```

You can easily modify this to hash a section of memory.

How to add junk code to my project?

You add code that essentially does nothing important. You must disable optimization as the compiler will know the code isn't important and remove it.

Limitations of junk code

Junk code essentially changes the file hash and the hash of the code sections that contain it. It does not defeat signature detection, which utilizes pattern scanning to compare every byte in memory against the signature you have built to identify that binary.

Note that if an anticheat is detecting you based on file hash and section hash, you can add junk code and your hack may no longer be detected. But they then can hash your new hack and detect it again. It's only a temporary solution.

Adding junk code will not bypass signature detection -> You must use polymorphic code

The solution to these limitations is to use polymorphic code.

What is polymorphic code?

Polymorphic code changes the assembly of your binary every time it's loaded into memory. Or perhaps it only needs to be done on a per-user basis, for example if you're distributing a pay hack. Or perhaps you only want to polymorph it every couple of days/weeks to keep anticheat off your ass.

High tech malware often uses polymorphic code to bypass antivirus but more often you see the usage of Crypters that obfuscate the assembly on disk, but when it is loaded into memory it's usually the same. That is why antivirus scans memory and files on disk.

Polymorphic code defeats signature based detection of your CODE. Things like strings, PDB location or other parts of DATA sections can still be used in signature scanning.
Your poly implementation must change almost every byte of code, I would say every 5 bytes at a minimum. When you make hacks using pattern scanning you will see the average pattern size is maybe 10 bytes and some of them are wildcards. You do not know what bytes the anticheat is using to identify your hacks so you must over do it.

Who needs to use polymorphic code?

Pay cheat providers. No one else needs to use this. Anticheat developers do not have time to build signatures for every single cheat out there but they do build signatures for paid cheats that do not have protection. They'd have to flag your

code as suspicious and send it to their team for analysis. Obviously they build sigs for all the public hacks you can download, if you wrote a polymorphic loader you could technically polymorph a detected hack and make it undetected as long as code signature was the only method they use for identification.

There are much easier ways to bypass anticheat, you can simply patch or hide from most.

Examples of Polymorphing code as could be created with a Junk Code Generator

Let's say the anticheat built a signature for this assembly:

C++:

```
mov ecx, 5
sub  ecx, 5
add ecx, 5
```

Here are some examples of what a polymorphed version might look like

C++:

```
mov ecx, 5
push eax
pop eax
sub  ecx, 5
add eax, 5
sub eax, 5
add ecx, 5
```

C++:

```
nop
mov ecx, 5
nop
sub  ecx, 5
nop
add ecx, 5
nop
```

C++:

```
push edx
mov edx, 5
mov ecx, edx
pop edx
sub  ecx, 2
sub ecx, 3
push edx
mov edx, 5
add ecx, edx
pop edx
```

Notice the outcome of the instructions is identical, but the signatures won't match. The only thing limiting methods of polymorphism is your imagination, the secret is to do it with the least number of additional instructions, so your code is still efficient. My personal favorite is the NOP in between every byte, would be simple to implement and funny. Ideally you would not modify the number of instructions. If your polymorph engine adds bytes rather than replacing bytes, you must resolve all relative and hard coded addresses. It is much easier to modify the assembly without adding additional bytes.

Also if you make a payhack you would want to use several different algorithms, as heuristics could easily detect just one variant. Again tho, there are much easier ways to bypass anticheat then polymorphism.

So how do you polymorph your hack? Here are some techniques

At compile time using a modified compiler

Like Chris Domas who wrote his own compiler that only uses the MOV instruction

When you compile your project a bunch of .obj object files are made, you could use some method to polymorph them during the linking stage of compilation

At runtime or Before injection

Make your hack.exe read hack.exe from disk, polymorph it, write it to disk and then run it
Make your injector.exe polymorph hack.dll, write it to disk then inject it

During Manual Mapping

modify the assembly during your manual mapping routine, if you don't know what that is, here is our guide on manual mapping:

https://youtu.be/qzZTXcBu3cE

There are 2 more polymorphic posts below this one:
mambda's and Liduen's

Additional Resources

- Metame - Metame Is A Metamorphic Code Engine For Arbitrary Executables
- Replacing common x86 instructions with less known ones
- x86 Disassembly/Code Obfuscation - Wikibooks, open books for an open world
- andrivet/ADVobfuscator
- https://www.esat.kuleuven.be/cosic/publications/thesis-199.pdf
- https://www.defcon.org/images/defco.../defcon-17-sean_taylor-binary_obfuscation.pdf
- Replacing common x86 instructions with less known ones
- Tutorial - Random number generator at COMPILE TIME

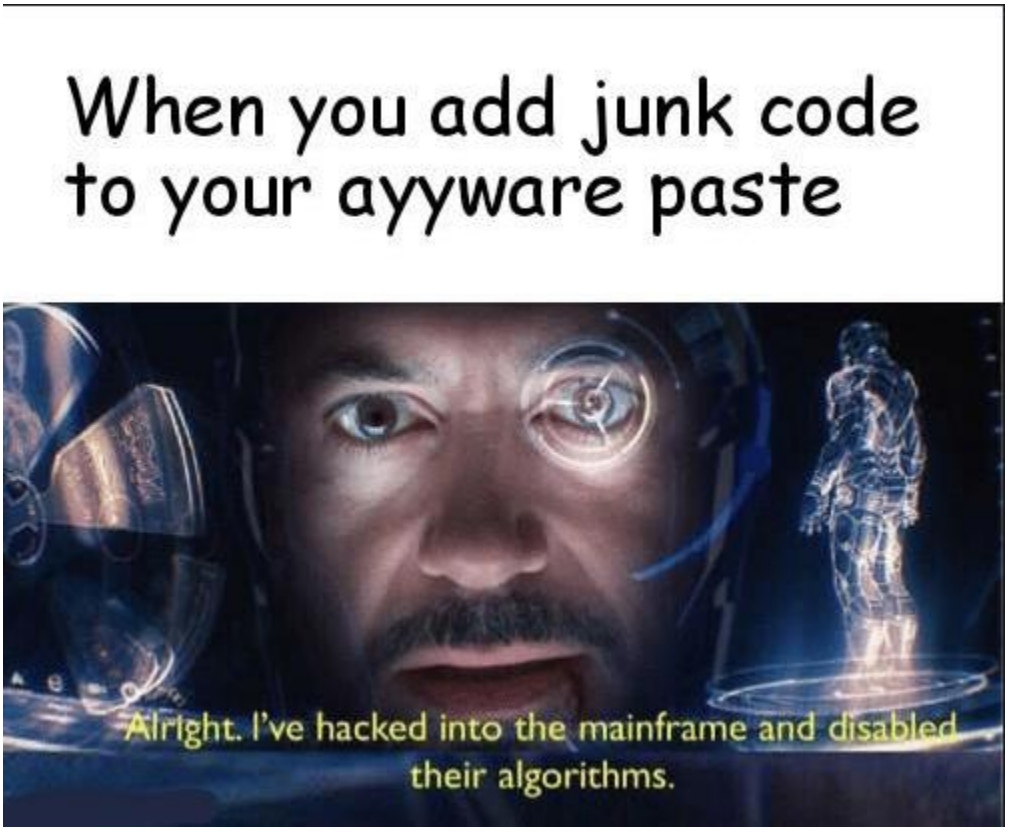
How To Bypass VAC Valve Anti Cheat Info

VAC or Valve AntiCheat is software running on the client and server that attempts to detect cheaters. It is made by Valve and has been around since the early days of Counter Strike, most known for it's usage in CSGO but, is also used in other Source Engine games. VAC is a usermode anticheat, it does not have a kernel mode driver, It's primary detection mechanism is signature scanning for known cheats. Here you will find a list of compiled information from the forum about how Valve Anticheat works and how you can bypass VAC.

This anticheat guide features:

- An explanation of the new VAC updates from 2020 and 2022
- The 5 simple steps you need to take to bypass VAC
- A brief overview of VAC's modules
- A more in depth look at VAC's capabilities
- A collection of VAC related resources

Before you read this VAC Guide, you may want to read our general overview of how anticheat works -> General Anticheat Guide



We answer the same 3 questions about VAC at least once per week, please just read this information instead of annoying us.

Important VAC Update July 2020

Valve has been actively updating VAC in CSGO this month. Over the years new competitive shooters like Apex, Overwatch & others have been released with either strong anticheat or kernel anticheat and these games have fewer cheaters than CSGO due to VAC being worthless. Now Valorant which is very similar to CSGO has been released with a very good kernel anticheat. 30% of CSGO players are cheating, and now that alternatives are available people are leaving

CSGO. This has forced Valve to improve VAC, this month some of the largest changes that have ever happened to VAC are being rolled out and I assume more will come soon.

CSGO is now starting in Trusted Mode by default
Use the -insecure launch argument to practice and develop your hack in a local bot match. After being sure you are able to bypass VAC, launch with Trusted Mode later.

CSGO is now blocking DLLs from being injected using LoadLibrary - DLLs that interact with CSGO must now be digitally signed
To bypass this all you need to do is use Manual Mapping - try the GH Injector's special features.

Valve Anticheat Update February 2022

VAC now scans your cvars for anomalies, this includes m_flFlashMaxAlpha, m_bSpotted & m_clrRender which are commonly modified by cheats. To bypass this, you would have to hook the cvar scanner and spoof the results. VAC is rather easy to reverse but your average game hacker isn't smart enough to do it. @dretax pointed us to this information here: VAC detection · Issue #109 · dretax/GarHal_CSGO
Not sure if this information is 100% accurate but I think everyone can agree VAC is getting more serious lately.

How to bypass VAC

Warning: Things have changed in the past 2 years. VAC has been getting more aggressive.

VAC doesn't really ban unless you're doing something really bad. In most cases just your Trust Factor will be lowered and you'll only be playing with other cheaters.

This was our original instructions from 2 years ago:

It's really easy. You do not need to ask us how to bypass it. Just read these few paragraphs and you'll be bypassing VAC in 5 minutes.

- Manually Map your DLL
- Do not use public downloads and source codes
- Write everything yourself, do not share your hack
- Do not use VMT Hooking, use a regular detour / trampoline hook, or better yet a midfunction hook (not at first byte of function)
- Don't rage, use human like features

If you do these things, the chance that you will get VAC banned is less than 1%. By doing these things you have bypassed 99% of VAC. You can never be 100% safe so don't even worry about it.

VAC is honestly a joke, if you're just learning how to hack don't worry about VAC. Just learn how to hack and write cheats for CSGO, if you get banned just create a new account, the game is free. Stop asking "how to bypass VAC" it's the dumbest question. All you have to do is follow the steps written above.

If you enjoy the content you find here on GH, please considering donating.

2020-2022 Valve Anticheat Bypass Updates

Like we've said, VAC has become more aggressive lately. You can still hack this game for fun, but you'll probably get reduced Trust Factor.

How to 100% bypass VAC?

1. Do everything listed above.
2. 100% understand this: GitHub - danielkrupinski/VAC: Source code of Valve Anti-Cheat obtained from disassembly of compiled modules
3. Learn how to dump VAC modules and reverse them yourself
4. Make a complete VAC bypass by hooking all of their scans/modules

I'm not teaching you how to do it, if you just complete the GHB, this is really simple to do.

VAC Detects VMT Hooking

There is a good amount of evidence that VAC detects VMT hooking, to bypass this just use a regular detour/trampoline hook. Or if you want to be extra safe, do a mid function hook (regular detour, not located at the first byte of the function) so you're not easily detected by checking the first byte of the function.

Is WriteProcessMemory detected?

Everything is detectable, the real question is: will you get banned for using it? No you won't, so just use it and stop asking.

Is _____ detected by VAC?

VAC is actively scanning all your running processes, files, registry keys & more. If they want to know everything that's happening in usermode, they have no problem doing it. Everything in usermode can be detected by VAC. It doesn't matter if VAC is capable of detecting something. The only thing that matters is: are they banning people for it.

Do I need to use kernel mode to bypass VAC?

2020 Answer:

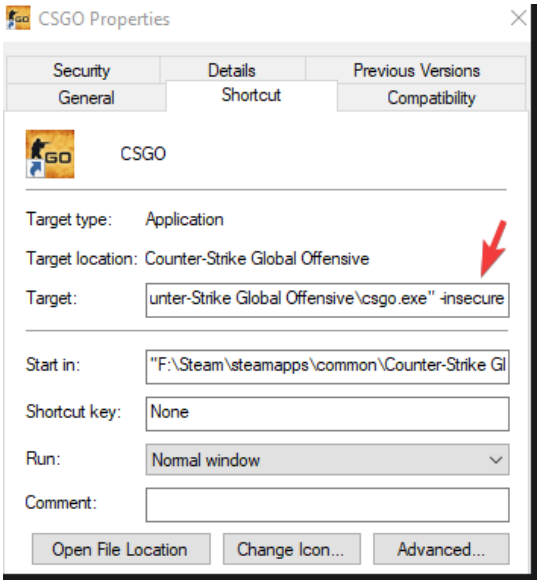
NO! VAC is a usermode anticheat. There is no reason to go into kernel unless you want to. It's complete overkill.

2022 Answer:

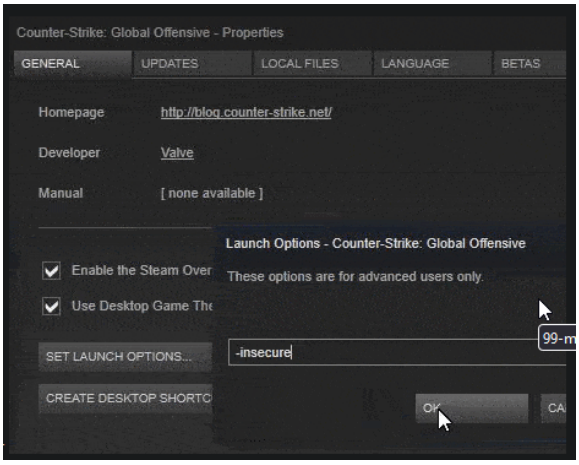
VAC is getting more aggressive lately, using a driver to read and write is probably a good idea now. Also use it to hide your hack and hook VAC itself and spoof the result of all it's scans. If you just want to learn how to hack and cheat for a couple weeks, no you don't need to go kernel.

Insecure Mode

The first thing you must do when creating hacks is to set the game in insecure mode. This is done by adding the "-insecure" command line option to your desktop shortcut. Once this is done you can develop your hack or use Cheat Engine on the game without worrying about being banned. In insecure mode you cannot join secure servers.



or



How to bypass VAC?

There is no magic trick or download we can give you to instantly bypass anticheat. If you have been game hacking for less than 6 months, you have no business asking about anticheat. You cannot even understand because you do not have the required knowledge to do so. Learn how to hack first for a few months before even thinking about bypassing anticheat, you can learn everything from The Game Hacking Bible.

Here's a great quote from @c5 regarding VAC:
The issue with incapacitating VAC are its heuristics and diversity of checks. It does a lot of cross checking, lies on different techniques on achieving the same task, etc. Besides, some things are only triggered when a specific flag is raised, so even if you might think you have bypassed or caught some of its methods in action, another path can be taken and your efforts countered.

At the end of the day though, you can lie to, emulate or disable anything that's running on your PC. People have emulated anticheats before, disabled them, altered scan results, hidden cheats from them, etc. It can simply get very tedious and not worth the time at all, especially if all you want to actually do is simply bunnyhop around the map.

c5 is 100% right. If you're just making cheats for yourself like the other 100,000 that are doing so, there is nothing to worry about, VAC is a joke. But it does have the capability to do much more than they use it for.

VAC's Capabilities

While VAC is loaded it has the capability of and has been seen:

- Scanning all your files
- Scanning all running processes
- Scanning your registry
- Enumerating all open handles
- Scanning for hooks
- Signature scanning for known cheats

With these capabilities it can find and detect cheats very easily.

Valve Anticheat does it basic run of the mill scanning on every client. But, if it finds something that looks sketchy like a hook, it will do a more thorough analysis and it will upload what it finds to the Valve servers. This information can have an affect on your Trusted rank or result in a ban in the future.

VAC Modules

VAC's modules are streamed to the client from the server, they don't exist on disk on your computer at any time but you can dump them if you know how. You can look at VAC as a series of module or as lists of features organized by purpose. The best resources for understanding VAC in depth are:

- Daniel Krupinski's Reversed VAC Source Code
- mamba's writeup of VAC from a few years ago
- Tutorial - RaptorFactor Archive - VAC Modules
- ioncodes/vacation3-emu

Vac's Modules according to Daniel Krupinski

- **Module 1: Collect System Information & Configuration**
- Module 2: Enumerate running processes and handles
- **Module 3: VAC's Process Monitor Implementation**

If you're reversing VAC yourself, make sure to look at steamclient.dll, SteamService.exe & steamservice.dll as well. VAC scans all 3 of those as well, so hooking those can be detected.

Advanced VAC Bypassing

If you're distributing a pay cheat you will want to reverse VAC yourself and periodically dump the modules and compare. If VAC updates, you need to know what they changed.

If you're distributing a pay cheat, in addition to our list above, you should:

- Encrypt all strings
- Randomize module, process, window & window class names
- Use polymorphic code to evade signature detection
- Stay off the disk as much as possible, stream everything into memory
- Clean all your tracks, avoid registry keys etc...

- Consider hooking and completely spoofing all VAC scans

It really depends, if you have 30 users you don't need to go too crazy. But if you have hundreds or thousands you need to be 100% sure you have bypassed VAC.

How does VAC protect itself?

VAC modules are streamed from the server, it does not hit disk. IAT is encrypted, strings are encrypted

VAC Detection Mechanisms

Signature Detection

Using various heuristics VAC can find suspicious code and upload the modules to their server for manual or automatic analysis. VAC doesn't have time to analyze every single cheat, they prioritize cheats that are used by many clients, the less people using it the less likely they will build signatures for it. They build signatures for the code, just like we do when pattern scanning or AOB scanning in Cheat Engine. VAC can use any part of your hack to build unique signatures including file hash, strings, PE header, window titles, PDB path's etc...

They scan the game's process as well as any other running process for these signatures, if the signature is found they know you're cheating and can ban you in the next ban wave.

VAC uses VirtualQuery() to find executable memory and scan the game process for memory pages that are executable, if these pages were not allocated by the game process it's obvious this is injected code and maybe a cheat. That's the first step to VAC sig scanning, it's gotta find the executable memory first because code makes the best unique signatures.

Hook Detection

VAC can detect all hooks, but we know they are very ban happy when it comes to VMT & IAT hooks.
They specifically scan for hooks in these Windows API functions:

- GetMAppedFileNameA
- NtQueryVirtualMemory
- GetModuleHandleA
- GetModuleFileNameA
- OpenProcess
- ReadProcessMemory
- VirtualQuery
- VirtualQueryEx
- CreateToolHelp32Snapshot
- Module32First
- Module32Next
- Process32First
- Process32Next
- EnumnProcessModules
- GetModuleBaseNameA
- GetModuleFileNameExA
- EnumProcesses
- GetModuleHandleExA
- GetMappedFileNameA
- NtReadVirtualMemory
- NtQueryVirtualMemory
- NtMapViewOfSection
- NtOpenProcess
- NtQuerySystemInformation

If a hook is detected, it will find the module where the jmp redirects too and send that data to the server for analysis or ban.

File Integrity Checks
All hacks must be done at runtime, important files are checked for integrity. Patching the files on disk is a no no.

VAC Enumerates all running Processes
VAC uses EnumProcesses to find all processes and does further scanning of these processes. This is the beginning of it's external hack process detection. Hiding your external hacks and injectors from EnumProcesses is the first step. They can't build sigs for something they can't see right?

EnumWindows & EnumChildWindows & GetWindowText
If you have a suspicious external process they will find the windows associated with them and get the window title. They make a hash of your window names and compare against known cheat window names. They also grab the window style, size & location which makes for easy external overlay detection. Maybe make your overlay larger than the game window and then offset all your drawing to the right position. Making your overlay the exact size of the game window is a dead giveaway it's an overlay cheat.

File Hashing
VAC creates files hashes for all running files or files recently touched by the OS and compares it against known cheat file hashes.
You can easily change file hash by simply adding bytes at the end of the file with any hex editor, of course you can automate that. This only prevents file hash signature detection.

VAC calls NtQueryInformationProcess()
Using ProcessBasicInformation it gets the address of the PEB. Using the PEB is the lowest usermode way of querying a process, by doing this it bypasses any patching/hiding you've done to other higher level documented APIs.

NtFsControlFile() & USN Change Journals
VAC scans the disk for every file that has recently been touched by the operating system, including deleting, renaming,

creation & overwriting. Good luck hiding from that
To bypass this mambda suggests hooking NtFsControlFile()

Manual Mapping
Manual Mapping defeats many module detection methods that VAC and other anticheat have such as:

1. LoadLibrary hooks
2. Toolhelp32Snapshot
3. EnumprocessModules to find loaded modules
4. Walking the PEB loaded modules list
5. GetMappedFileName() on memory addresses to find DLL's on disk

Misc things Valve Anti Cheat does

- Easily detects debuggers but doesn't prevent them
- ntdll.dll is scanned, patching functions in here will lead to detection
- VAC uses EnumDeviceInterfaces() to find all drivers in device manager
- Reads the Event Log for recent events such as driver loading
- Reads the registry

New Machine Learning in VAC

VacNet: Server Side Machine Learning to find cheaters based on statistics.

https://youtu.be/kTiP0zKF9bc

How VAC Bans Work

Valve AntiCheat bans in waves usually, you could be banned hours, days weeks or months after using a detected cheat. If it's a public cheat, you can guarantee you will get VAC banned if you use it after they build signatures for it which only takes maybe a week or 2 in most cases. If you haven't been banned within 4 weeks you're probably okay.

VAC doesn't do IP or HWID bans. Every time someone gets banned, they buy a new account, making Valve tons of money so they will never do this. If you get banned, make a new steam account. But HWID and IP are used for Trust Factor, if they detect a new account from a computer with multiple bans, your trust factor will be penalized.

Junk Code / Polymorphic Code

Adding junk code to your hack will change the file hash, and avoid detection based on file hash. You can also simply do this by adding bytes to the end of the file. But VAC also hashes the code sections, so junk at the end of the file won't work, but adding junk code will actually solve this problem. Junk code is just code that does nothing in your hack, you can put any code you want in there as long as it doesn't modify the functionality of the hack logic.

BUT adding a few pieces of junk code will not bypass signature detection, only hashing.

You need to use polymorphism to bypass signature detection. Polymorphism will change the assembly at almost every byte, ruining all possible signatures. Read our guide on polymorphic code here
Or just completely bypass VAC so it can't even sig scan you.

CSGO Overwatch

Overwatch is a crowd sourced moderation system, if you get too many reports, demos of your gameplay will be reviewed by other players. If the majority of other players file their Overwatch reports with the opinion that you are violating the rules, your overwatch reputation will decrease and it will eventually result in a ban.

CSGO Match Making & Trust Factor

Griefers and cheaters will have a lower trust factor, this is based on many things including Overwatch reports. Match Making matches people with high trust factor with other similar players. Conversely it puts cheaters and other people with low trust factor in the same matches.

Trust Factor is tied to HWID/IP, if you get banned and make a new account, some of your old Trust Factor will make it to your new account.

Learn more about Overwatch, Match Making & Trust Factor: #1, #2 & #3

Additional GH VAC Resources

- mambda's Original VAC Writeup
- c5's VAC Reverse Engineering IDA Scripts

Offsite VAC Resources

- VAC Source Code
- Developments | Cra0kalo's Development Adventures
- zyhp/vac3_inhibitor
- danielkrupinski/VAC-Bypass

Continue reading the rest of the thread for more info...

Thank you to the contributors to this guide:
@mambda @XdarionX @KF1337 @ZleMyzteX

Disclaimer: Information I'm spewing is from reversal that happened in 2015, more in depth information can be found at: RaptorFactor.com for example, however it seems he no longer wants to update that for the time being.

How does VAC detect things?

Well, there are a few methods that it uses in order to flag things, but the main method of detection when it comes to VAC is signature based detection (henceforth known as SBD.).

It's quite simple, you compile something and the resulting binary is a series of bytes, say your ultra leet cheat has the bytes 37 13 37 13 37 13 37 13 all in order, and its only used in one specific place all the time, and that place has the bytes, say, 0x6A <offset to the 37 13 shit above> .

That's something that could potentially be used as a signature. In essence, a signature is simply a pattern that can be found in a binary, preferably something that will be exclusive to that binary, this can be anything from a specific byte sequence in instructions, a specific string, pdb data, etc.

So valve basically hashes various portions of a binary that it deems suspicious, and checks the resulting hash with a few other hashes it has stored to decide whether or not something is a known cheating software, in which case, you get flagged and will get the hammer later.

Of course, it's not the only thing that valve does, they also, for example, enumerate all top level windows and hash things such as the window name, some attributes (i.e. transparent iirc.), position and size (basically checking for overlays on top of the game).

It's also got some more cool shenanigans, you can read more about some of its external related things here :

It is to be noted that valve does much more than *just* look at simple bytes in your program, and just because you have a driver doesn't mean you're 100% vac safe. get the binaries and reverse them and everything is clear and all that shit.

VAC

- Loads many modules during games.
- When something attempts to debug (or open a handle ?) to steamservice.exe it is immediately checked out
- It doesn't seem to care about anything on community servers, but definitely cares on casual & competitive
- In some module it gets the main drive ("C") and recursively queries directories that aren't Program Files (? maybe ?) cheat folder enumeration
- On startup SteamService.exe checks SteamService.dll for file integrity, aka no patching on disk.
- Look like searches for Clear Information/FilterManager in Event Logs?
- OpenEventLog("System")

- ClearEventLog(givenHandle);

So how do we make our cheat bypass VAC?

- Have the cheat start before csgo.exe starts
- The cheat first injects the dll, then protects itself and demotes the privelages of steamservice.exe
- Then you run csgo.

Successful Reversed Modules

7C34.tmp

- GetNativeSystemInfo() - returns a pretty useless struct for me to care about.
- NtQuerySystemInformation [TimeOfDay, CodeIntegrity, DeviceInformation, KernelDebugger, BootEnvironment, RangeStart]
- Reads some important parts of NtDll.
- Does various checks

SteamService.exe

- On game launch and steamservice.exe startup, SteamService.exe calls EnumProcesses with a size of 4096 (aka 4096 / 4 is the count of processes) to get all running processes.
- Creates a file mapping on startup. format: "Steam_{E9FD3C51-9B58-4DA0-962C-734882B19273}_Pid:%000008X", steamServicePID
- Some event triggers telling csgo vac system is being blocked: i know this can happen due to USN being cleared, but could our VQEx hook also do it?
- VAC communicates with Pipes. cool stuff, need to research those more.

63CE.tmp - Internal(?) Module

- at some point it calls VirtualAlloc() on its own process with size 18016d , MEM_COMMIT, PAGE_READWRITE
- Later on it queries the process with NtQueryInformationProcess for **ProcessBasicInformation** , if this fails to get a buffer of size 24 it returns with 60;
- If successful, continues on with ImageFIlename
- It reads lots of predetermined memory regions. It uses VirtualAlloc on its own memory possibly for further inspection by host process.
- Also reads to csgo memory
- Lots of calls to VirtualAlloc()

Basically this guy opens specified process ID & does some VirtualQueryEx, I believe this checks for whether there is executable code in the csgo.exe module.
Checks queried memory for protect flag and Allocation Protection 0xF0
0xF0 = (PAGE_EXECUTE | PAGE_EXECUTE_READ | PAGE_EXECUTE_READWRITE | PAGE_EXECUTE_WRITECOPY)

If neither of these are found, v10 = 1

Could this read be doing sig scanning being that it reads information? I wonder if any of these open the file mapping.
v10 is placed in a2 + 60 , so definitely return value.

A2 Struct
=====
a2 + 56 = LastError()
a2 + 60 = returnValue
=====

441F.tmp - Device Module

Enumerates hardware devices with Setup Api.dll
EnumDeviceInterfaces to be exact. literally ALL OF THEM FROM DEVICE MANAGER AND PROBABLY BEYOND LOL.
Thats basically it. Underwhelming tbh.

FAF2.tmp - Volume Module

- Begins to search all volumes with FindFirstVolumeW, FindNextVolumeW and closes handles with FindVolumeClose
- Gets volume serial with GetVolumeInformationW and checks if it matches a predetermined serial
- I assume this is the volume serial hash.
- Gets a specific process' name and reads its memory (i presume this is csgo).
- Also does this with another process where a handle is given. instead of a pid.
- Another seciton where they GetMappedFileName

Aha! EnumProcesses!
Opens a process to every handle running with query_information and vm_read , tries to get their name and do some more things that i can't see yet.

Course of action here for my external : Strip handles of those values ^ , i don't really care about anything else. They can't get my name if they dont have the privileges to. Also they couldn't find it on file if they tried.

F335.tmp - Window Module

- EnumWindows finds ALL top level windows (overlays too) , also does EnumChildWindows.
- They enumerate your windows and if your process id == something that they have stored then they will GetWindowInfo your
- They will keep your style (and exStyle), WindowStatus, WindowBorders (x and y)
- It then calls GetWindowTextA and a secondary function
- for most externals exStyle = WS_EX_TOPMOST | WS_EX_TRANSPARENT | WS_EX_LAYERED
- Then i got lazy because there was a huge function up next, probably hashing.
- Basically, if your PID is something that it's looking for (specified by parameters), it will try to enumerate your window and log all those things ^ & probably send them back
- In the end they make a hash of your window name (from GetWindowTextW)
- They compare these with various hashes (13 to be exact)

7B0B.tmp - File Mapping Module

FileMapping module, for now it seems to be majorly worthless, but there are some indirect function calls that i cant seem to pin down to figure out what its doing to the file mapping.
However it only gets opened with read permissions so i doubt its anything major.

BAC1.tmp - USN Module - Update Sequence Number Journal

GetVolumeInformation
This is later used with NtFsControlFile with FSCTL_QUERY_USN_JOURNAL
You get UsnJournalData via DeviceIoControl (they use the higher up NtFsControlFile) , it returns a USN_JOURNAL_DATA struct .
So you set whatever you want (i.e. READ_USN_JOURNAL_DATA struct) 's id to whatever the journal id is
Alright, after some painstaking hours i managed to reproduce their usn querying.
if USN Region matches these flags: USN_REASON_CLOSE | USN_REASON_STREAM_CHANGE | USN_REASON_REPARSE_POINT_CHANGE | USN_REASON_RENAME_NEW_NAME | USN_REASON_RENAME_OLD_NAME | USN_REASON_FILE_DELETE | USN_REASON_FILE_CREATE | USN_REASON_NAMED_DATA_TRUNCATION | USN_REASON_NAMED_DATA_EXTEND | USN_REASON_NAMED_DATA_OVERWRITE
In laymans terms this means : If the file has recently been closed, created, deleted, renamed , or overwritten/written to, we want to check that out.
Then we hash the partial file name and reaon flag and compare them to some hashes
This happens with various other parts of the usn struct

NtFsControlFile
They do a crapton. The best thing to do is hook NtFsControlFile after it returns from KM and then clean any references to my stuff.

Here's what I can think of for this:

- IAT Hook NtFsControlFile and redirect it to my own function with the original address stored.
- Call the original function.
- parse the allocated memory for any data regarding my own stuff, if found, purge it.
- return.

So you can't IAT hook something you have to GPA, past me
So we hooked GPA via IAT (so no modified bytes here)
from that, we check for when GPA is called for NtFsControlFile and we instead return the address of our own function while saving the actual location.
In our function we (setup stack b*tch) call the original, then check if the control code was FSCTL_READ_USN_JOURNAL. if it was, we check out the USN_RECORD and check if the filename contains 'SPQR' , if it does, then we purge it and continue as normal.

69D7.tmp - Event Log Module

=======

- Pretty funky encryption here
- Goes through the event log with OpenEventLog, ReadEventLogA, EvtQuery, EvtCreateRenderContext (for system and user information)
- Enumerating newest things first
- So I think I want to load my driver then clear the event log

C022.tmp - Registry Module

=======
Didn't look too far into this one, seems to enumerate registry keys (possibly to detect drivers or certain p2cs ?)

BBC7.tmp - Majorly worthless, File mapping stuff.

=======

{%02xDEDF05-86E9-%02x17-9E36-1D94%02x334D-FA3%2xA0441} is used as format for opening a file mapping.

991E.tmp - SysEnter module

- Manually calls sysenter with the ordinal passed into it by SteamSerive.exe/dll , funky stuff.
- Calls EnumprocessModules
- Gets module base name and information

CEA4.tmp - VirtualQuery Module

- Calls VirtualQueryEx on specified regions.
- If the type is MEM_FREE it breaks and basically exits.
- on MEM_RESERVE it increments region size, possibly to try again and also sets a variable to true
- MEM_COMMIT it does checks to see whether the page is executable (0xF0) and if so it logs that and increments some values
- More interestingly, this module gets file names using GetMappedFileName and it opens the file with read access.
- It reads the file in its entirety and updates an MD5 hash with the bytes.
- Dat public cheat detection tho.
- Manual mapping itself fixes this because they won't know the file name to read it on disk.

steamclient.dll

There is something in here that logs where every injected file is in memory and writes it to a section

{%02x3F1461-5E%02x-4E99-A5AE-CEFDB55A%02x2D-3DED%02x3C}
format = pid >> 8, pid >> 24, (pid >> 16) & 0xFF, (unsigned __int8)pid
We open this with READ_WRITE permissions, we check for our string, if we find it, we zap away the entirety of it from the section
section struct size seems to be 0x4F (79 dec)

There's also another global handle that logs open handles

Okay so: On DLL_THREAD_ATTACH vac queries the memory and does a few scans, checks for some flags that are retarded:
Gets the moduleFileName
"If something suspicious is found, VAC uses the first module to analyze it. I didn't look into the first module, but it extracts the image sections does tons of hashes, maybe something more."

To circumvent this you need to manual map.

E2D5 - Sig Scanner

- Calls VQueryEx, RPM, like all vac modules. (RPM that is, not vqx)
- If the return value is not >= 0x1C then it skips all the funky stuff that could be sig scans.
- Allocates memory after initialization, 0x10000 bytes MEM_COMMIT | MEM_RESERVE , PAGE_READWRITE
- this memory is where the final RPM is placed which they then attempt to hash and compare

Yeah im done with this now. Get Fukt valve.

Fun Facts:
Seems every module has the ability to get your volume serial, gotta be sure amirite vac? haha

Plan of action:
Externals : Hook K32Enumprocesses, hide my pid.
Internals : Hook VirtualQueryEx , when they query my memory tell them its non-executable so they bugger off, maybe even hook K32EnumProcessModules if they call it on csgo.exe...
MANUAL MAP BOYS.

Anticheat Battleye Bypass Overview

BE is the second most popular mature, kernel mode anticheat. Battleye does many of the same things as EAC but it is less popular and easier to bypass. Despite being easier, you still need to know what you're doing if you want to start hacking a BE protected game. This article will tell you everything you need to get started with a Battleye bypass.

We have two guides which should be viewed before reading this BE specific guide:
Guide - How to Bypass Anticheat - Start Here Beginner's Guide
Guide - How to Bypass Kernel Anticheat & Develop Drivers

This article contains information compiled from many sources, full credits to these gentlemen: @iPower, @_xeroxz, vmcall & everyone at secret.club

(img courtesy of BattlEye – The Anti-Cheese Gold Standard)

Games utilizing Battleye

- Fortnite
- PUBG
- Escape from Tarkov
- Rainbow Six Siege
- Ark Survival Evolved
- ARMA II
- ARMA III
- DAYZ
- H1Z1
- Surivial Of the Fittest
- PlanetSide 2
- Survarium
- Project Argo
- Unturned
- Insurgency
- Day of Infamy
- The Isle
- Line of Sight
- Conan Exiles
- Tibia
- Black Squad
- S4League
- Zula
- Islands of Nyne
- BlackLight Retribution
- SOS
- Pixark
- Heroes & Generals
- Bless Online
- and more

Battleye Anticheat Versions

It's important to understand that the version of BE is not the same on every game, on an older game it will be easier to bypass. Newer more popular games will have the latest version. Battleye has been around since 2004 and has been actively developed throughout it's history. Tutorials and information from 5 years ago will not work on the newest versions, but still worth reading. Battleye was first developed as a third party anticheat for Battlefield Vietnam and Battlefield 1942 but became more popular and robust with it's integration with ARMA 3 and DayZ.

Battleye Anticheat is a Kernel Mode Anticheat

A processor in a Windows computer has two different modes: *kernel mode* and user *mode*. The Usermode & Kernelmode construct is built into the CPU. The low level core functionality of the operating system is done in kernel mode, which is a privileged part of memory that is not accessible from user mode and executes with privileged status on the CPU. Drivers are not just limited to Hardware Drivers, you can make a .sys driver to do anything you want in kernel mode, including bypass anticheat and perform cheat functionality. Usermode and kernel are separated, nothing you do in usermode will bypass the kernel driver.

Because BE is a kernel mode anticheat you will also need to be in kernel to make a Battleye bypass.. You can use a VM or hypervisor to dump the Battleye module and reverse engineer it, keep in mind BE does have some emulation detection.

Read the main Kernel Guide to learn everything you need to do know before you start working on Battleye.

But Rake, I don't want to learn, I just want to paste a Battleye bypass!

Ok before we go to far I will give you a simple 6 step process that is the easiest way to paste your way into kernel:

1. How to Make a Windows Kernel Mode Driver Tutorial
2. Kernel 2 - Usermode Communication - IOCTL Tutorial
3. How to Write Memory from Kernel - MmCopyVirtualMemory Tutorial
4. Experiment with this source code Source Code - CSGO Kernel Driver Multihack
5. Use kdmapper which uses a vulnerable Intel driver to manually map your kernel driver (make sure anticheat is not loaded yet)
6. Start the game and use your usermode application to write to the game memory

With those 6 steps, you can start reading and writing to a BE protected process. Battleye and other strong kernel anticheats can detect this easily, so keep reading to learn how to stay undetected. You haven't bypassed the actual Battleye detections with this, you're just giving yourself the ability to read and write, which you should use to dump the Battleye modules. Using this method by itself will get you banned, keep reading.

Manually Mapped Driver Detection

To avoid your manually mapped driver getting detected you need to clear PiDDBCacheTable & MmUnloadedDrivers, and stop the enumeration of your own system pools & threads.

- PiDDBCacheTable & MmUnloadedDrivers
- system pool detection
- system thread detection

@iPower said they search for system threads which do not belong to any regular kernel module, easily detecting manually mapped drivers. You can find it in his logs by searching for PsLookupThreadByThreadId & RtlWalkFrameChain.

Battleye Anti Cheat Components

- **BEService** - Windows service that communicates with BEServer, which provides BEDaisy and BEClient communication capabilities
- **BEDaisy** - kernel driver that registers callbacks and minifilters to prevent cheaters from modifying the game
- **BEClient** - usermode DLL that is responsible for most of the detection vectors, it is mapped into the game process after initialization
- **BEServer** - backend-server that is responsible for collecting information and taking concrete actions against cheaters

Battleye Anticheat Features

- Debugger detection
- Signature based detection of known cheats
- Open game process handles
- Detection of manually mapped modules, i.e. executable pages not backed by a image on disk
- Process handle creation is blocked
- Overlays detection
- Steam Overlay hooks and hacks embedded in steam process's
- lsass.exe modifications
- game files integrity checks
- TCP connections to cheat sites
- module name and timestamp blacklist
- certificate blacklist
- driver blacklist
- stack walking / ret check
- single stepping to detect code outside of usermode memory range
- hypervisor-detection.html']hypervisor detection[/URL]

Battleye is actively scanning and uploading a lot of information to their servers while you play:

- all running processes
- all device drivers
- all window names
- options to upload more if anomalies are detected

How does Battleye protect itself?

- virtualization
- streams shellcode into memory
- integrity checks on it's modules & shellcode
- encrypted traffic with BE server
- encrypted named pipe communication
- it does extra logging on computers with lots of reversing tools

secret.club Battleye articles

secret.club has some of the best content regarding Battleye so you will definitely want to look at these

- BattlEye anti-cheat: analysis and mitigation
- BattlEye shellcode updates

- BattlEye stack walking
- BattlEye single stepping
- hypervisor-detection.html']BattlEye hypervisor detection[/URL]
- BattlEye communication hook
- Bypassing BattlEye from user-mode
- BattlEye reverse engineer tracking
- How anti-cheats detect system emulation
- How Escape from Tarkov ensures game integrity
- Cracking BattlEye packet encryption
- BattlEye client emulation

_xeroxz's Articles and Repos

@_xeroxz has done bunch of work on Battleye, on par with some of the secret.club articles, be sure to check them out too

- BadEye - BattlEye Handle Elevation Exploit
- xerox / BEDaisy
- xerox / badeye

Some important excerpts from his articles

BEDaisy Inline Hooks
BEDaisy places inline hooks on both NtWriteVirtualMemory and NtReadVirtualMemory inside of lsass.exe and csrss.exe. The reason for these hooks are because csrss.exe and lsass.exe need handles with PROCESS_VM_OPERATION in order to function properly. The handles that csrss.exe and lsass.exe would have to BEDaisy's protected processes are stripped of PROCESS_VM_OPERATION via BEDaisy's enumeration of the protected processes handle table by calling ExEnumHandleTable. In order to allow for csrss.exe and lsass.exe to read/write to the games memory BEDaisy proxies their read/write calls.

BEDaisy Imports
If you take a look at BEDaisy.sys's import address table you can see this nice little import by the name of MmGetSystemRoutineAddress, This function is used to dynamically resolve imports at runtime. List of BEDaisy imports: battleyes imports ($24) · Snippets

LOADED KERNEL MODULE ENUMERATION
BEDaisy enumerates all loaded modules by calling NtQuerySystemInformation with SystemModuleInformation. If a black listed driver is found, the game will not run, drivers like the notorious intel lan driver, capcom, and gdrv are all blocked by BEDaisy.

RUNNING PROCESSES ENUMERATION
BEDaisy also constantly enumorates running processes using NtQuerySystemInformation except with SystemProcessInformation, this can also be easily hooked to filter out specific executables from BEDaisy's queries.

ASYNCHRONOUS PROCEDURE CALL (APC)
BEDaisy registers APCs on all user mode threads in every process, the APC code that is executed simply calls RtlWalkFrameChain which inturn provides BEDaisy with all of the stack frames on the thread

== end of _xeroxz's content ==

ObRegisterCallbacks

Battleye blocks usermode access to a process by conventional means via ObRegisterCallbacks, essentially when you call OpenProcess() it will not let you get a handle to the game process so you can't read or write memory or attach a debugger. This was one of the first things implemented in Battleye. In order to circumvent that you need to hook their driver, collide with their callbacks, or simply remove their callbacks, read Douggem's article.

You can see it being called in @iPower 's log

Code:

```
[ LuluVisor ] TM -> KM Transition! Function called: ObRegisterCallbacks
[ LuluVisor ] Function called at: BEDaisy.sys+0028919c
```

In the past this was all that was needed to attach Cheat Engine to the game, but Battleye has been updated many times since this was implemented & it's protection has been improved over many years, just fixing ObRegisterCallbacks is no longer enough to bypass.

Bypass Process & Thread Callbacks

Here is a driver source code to disable the process and thread callbacks from anher0:

C++:

```
#include <ntifs.h>
#include <windef.h>

// Pre-Processor definitions for our I/O control codes.
#define REMOVE_BEOBJECT_CALLBACKS_IOCTL CTL_CODE(FILE_DEVICE_KS, 0x806, METHOD_BUFFERED,
FILE_READ_DATA | FILE_WRITE_DATA)
#define RESTORE_BEOBJECT_CALLBACKS_IOCTL CTL_CODE(FILE_DEVICE_KS, 0x807, METHOD_BUFFERED,
FILE_READ_DATA | FILE_WRITE_DATA)

// Global variable to our device.
PDEVICE_OBJECT deviceObj = NULL;

// QWORD
typedef unsigned __int64 QWORD;

// OLD_CALLBACKS
typedef struct _OLD_CALLBACKS {
    QWORD PreOperationProc;
    QWORD PostOperationProc;
    QWORD PreOperationThread;
    QWORD PostOperationThread;
} OLD_CALLBACKS, *POLD_CALLBACKS;

// CALLBACK_ENTRY
typedef struct _CALLBACK_ENTRY {
    WORD Version; // 0x0
    WORD OperationRegistrationCount; // 0x2
    DWORD unk1; // 0x4
    PVOID RegistrationContext; // 0x8
    UNICODE_STRING Altitude; // 0x10
} CALLBACK_ENTRY, *PCALLBACK_ENTRY; // header size: 0x20 (0x6C if you count the array
afterwards - this is only the header. The array of CALLBACK_ENTRY_ITEMs is useless.)

// CALLBACK_ENTRY_ITEM
typedef struct _CALLBACK_ENTRY_ITEM {
    LIST_ENTRY CallbackList; // 0x0
    OB_OPERATION Operations; // 0x10
    DWORD Active; // 0x14
```

```
    CALLBACK_ENTRY *CallbackEntry; // 0x18
    PVOID ObjectType; // 0x20
    POB_PRE_OPERATION_CALLBACK PreOperation; // 0x28
    POB_POST_OPERATION_CALLBACK PostOperation; // 0x30
    QWORD unk1; // 0x38
} CALLBACK_ENTRY_ITEM, *PCALLBACK_ENTRY_ITEM; // size: 0x40

// Dummy object callback functions.
OB_PREOP_CALLBACK_STATUS DummyObjectPreCallback(PVOID RegistrationContext,
POB_PRE_OPERATION_INFORMATION OperationInformation) {
    return(OB_PREOP_SUCCESS);
}
VOID DummyObjectPostCallback(PVOID RegistrationContext, POB_POST_OPERATION_INFORMATION
OperationInformation) {
    return;
}

QWORD GetCallbackListOffset(void) {
    POBJECT_TYPE procType = *PsProcessType;

    __try {
        if (procType && MmIsAddressValid((void*)procType)) {
            for (int i = 0xF8; i > 0; i -= 8) {
                QWORD first = *(QWORD*)((QWORD)procType + i), second =
*(QWORD*)((QWORD)procType + (i + 8));
                if (first && MmIsAddressValid((void*)first) && second &&
MmIsAddressValid((void*)second)) {
                    QWORD test1First = *(QWORD*)(first + 0x0), test1Second =
*(QWORD*)(first + 0x8);
                    if (test1First && MmIsAddressValid((void*)test1First) && test1Second
&& MmIsAddressValid((void*)test1Second)) {
                        QWORD testObjectType = *(QWORD*)(first + 0x20);
                        if (testObjectType == (QWORD)procType)
                            return((QWORD)i);
                    }
                }
            }
        }
    }
    __except (EXCEPTION_EXECUTE_HANDLER) {
        return(0);
    }
}

void DisableBEObjectCallbacks(POLD_CALLBACKS oldCallbacks) {
    POBJECT_TYPE procType = *PsProcessType;
    if (procType && MmIsAddressValid((void*)procType)) {
        __try {
            QWORD callbackListOffset = GetCallbackListOffset();
            if (callbackListOffset && MmIsAddressValid((void*)((QWORD)procType +
callbackListOffset))) {
                LIST_ENTRY *callbackList = (LIST_ENTRY*)((QWORD)procType +
callbackListOffset);
                if (callbackList->Flink && MmIsAddressValid((void*)callbackList->Flink))
{
                    CALLBACK_ENTRY_ITEM *firstCallback =
(CALLBACK_ENTRY_ITEM*)callbackList->Flink;
                    CALLBACK_ENTRY_ITEM *curCallback = firstCallback;

                    do {
                        // Make sure the callback is valid.
                        if (curCallback && MmIsAddressValid((void*)curCallback) &&
MmIsAddressValid((void*)curCallback->CallbackEntry)) {
```

```
                            ANSI_STRING altitudeAnsi = { 0 };
                            UNICODE_STRING altitudeUni = curCallback->CallbackEntry-
>Altitude;
                            RtlUnicodeStringToAnsiString(&altitudeAnsi, &altitudeUni, 1);

                            if (!strcmp(altitudeAnsi.Buffer, "363220")) { // Check if
this is BattlEye. If it is, disable the callback.
                                if (curCallback->PreOperation) {
                                    oldCallbacks->PreOperationProc = (QWORD)curCallback-
>PreOperation;
                                    curCallback->PreOperation = DummyObjectPreCallback;
                                }
                                if (curCallback->PostOperation) {
                                    oldCallbacks->PostOperationProc = (QWORD)curCallback-
>PostOperation;
                                    curCallback->PostOperation = DummyObjectPostCallback;
                                }
                                RtlFreeAnsiString(&altitudeAnsi);
                                break;
                            }

                            RtlFreeAnsiString(&altitudeAnsi);
                        }

                        // Get the next callback.
                        curCallback = curCallback->CallbackList.Flink;
                    } while (curCallback != firstCallback);
                }
            }
        }
        __except (EXCEPTION_EXECUTE_HANDLER) {
            return;
        }
    }

    POBJECT_TYPE threadType = *PsThreadType;
    if (threadType && MmIsAddressValid((void*)threadType)) {
        __try {
            QWORD callbackListOffset = GetCallbackListOffset();
            if (callbackListOffset && MmIsAddressValid((void*)((QWORD)threadType +
callbackListOffset))) {
                LIST_ENTRY *callbackList = (LIST_ENTRY*)((QWORD)threadType +
callbackListOffset);
                if (callbackList->Flink && MmIsAddressValid((void*)callbackList->Flink))
{
                    CALLBACK_ENTRY_ITEM *firstCallback =
(CALLBACK_ENTRY_ITEM*)callbackList->Flink;
                    CALLBACK_ENTRY_ITEM *curCallback = firstCallback;

                    do {
                        // Make sure the callback is valid.
                        if (curCallback && MmIsAddressValid((void*)curCallback) &&
MmIsAddressValid((void*)curCallback->CallbackEntry)) {
                            ANSI_STRING altitudeAnsi = { 0 };
                            UNICODE_STRING altitudeUni = curCallback->CallbackEntry-
>Altitude;
                            RtlUnicodeStringToAnsiString(&altitudeAnsi, &altitudeUni, 1);

                            if (!strcmp(altitudeAnsi.Buffer, "363220")) { // Check if
this is BattlEye. If it is, disable the callback.
                                if (curCallback->PreOperation) {
                                    oldCallbacks->PreOperationThread =
(QWORD)curCallback->PreOperation;
```

```
                                    curCallback->PreOperation = DummyObjectPreCallback;
                                }
                                if (curCallback->PostOperation) {
                                    oldCallbacks->PostOperationThread =
(QWORD)curCallback->PostOperation;
                                    curCallback->PostOperation = DummyObjectPostCallback;
                                }
                                RtlFreeAnsiString(&altitudeAnsi);
                                break;
                            }

                            RtlFreeAnsiString(&altitudeAnsi);
                        }

                        // Get the next callback.
                        curCallback = curCallback->CallbackList.Flink;
                    } while (curCallback != firstCallback);
                }
            }
        }
        __except (EXCEPTION_EXECUTE_HANDLER) {
            return;
        }
    }
}

void RestoreBEObjectCallbacks(POLD_CALLBACKS oldCallbacks) {
    POBJECT_TYPE procType = *PsProcessType;
    if (procType && MmIsAddressValid((void*)procType)) {
        __try {
            QWORD callbackListOffset = GetCallbackListOffset();
            if (callbackListOffset && MmIsAddressValid((void*)((QWORD)procType +
callbackListOffset))) {
                LIST_ENTRY *callbackList = (LIST_ENTRY*)((QWORD)procType +
callbackListOffset);
                if (callbackList->Flink && MmIsAddressValid((void*)callbackList->Flink))
{
                    CALLBACK_ENTRY_ITEM *firstCallback =
(CALLBACK_ENTRY_ITEM*)callbackList->Flink;
                    CALLBACK_ENTRY_ITEM *curCallback = firstCallback;

                    do {
                        // Make sure the callback is valid.
                        if (curCallback && MmIsAddressValid((void*)curCallback) &&
MmIsAddressValid((void*)curCallback->CallbackEntry)) {
                            ANSI_STRING altitudeAnsi = { 0 };
                            UNICODE_STRING altitudeUni = curCallback->CallbackEntry-
>Altitude;
                            RtlUnicodeStringToAnsiString(&altitudeAnsi, &altitudeUni, 1);

                            if (!strcmp(altitudeAnsi.Buffer, "363220")) { // Check if
this is BattlEye. If it is, restore the callback.
                                if (curCallback->PreOperation && oldCallbacks-
>PreOperationProc)
                                    curCallback->PreOperation =
(POB_PRE_OPERATION_CALLBACK)oldCallbacks->PreOperationProc;
                                if (curCallback->PostOperation && oldCallbacks-
>PostOperationProc)
                                    curCallback->PostOperation =
(POB_POST_OPERATION_CALLBACK)oldCallbacks->PostOperationProc;
                                RtlFreeAnsiString(&altitudeAnsi);
                                break;
                            }
```

```
                            RtlFreeAnsiString(&altitudeAnsi);
                        }

                        // Get the next callback.
                        curCallback = curCallback->CallbackList.Flink;
                    } while (curCallback != firstCallback);
                }
            }
        }
        __except (EXCEPTION_EXECUTE_HANDLER) {
            return;
        }
    }

    POBJECT_TYPE threadType = *PsThreadType;
    if (threadType && MmIsAddressValid((void*)threadType)) {
        __try {
            QWORD callbackListOffset = GetCallbackListOffset();
            if (callbackListOffset && MmIsAddressValid((void*)((QWORD)threadType +
callbackListOffset))) {
                LIST_ENTRY *callbackList = (LIST_ENTRY*)((QWORD)threadType +
callbackListOffset);
                if (callbackList->Flink && MmIsAddressValid((void*)callbackList->Flink))
{
                    CALLBACK_ENTRY_ITEM *firstCallback =
(CALLBACK_ENTRY_ITEM*)callbackList->Flink;
                    CALLBACK_ENTRY_ITEM *curCallback = firstCallback;

                    do {
                        // Make sure the callback is valid.
                        if (curCallback && MmIsAddressValid((void*)curCallback) &&
MmIsAddressValid((void*)curCallback->CallbackEntry)) {
                            ANSI_STRING altitudeAnsi = { 0 };
                            UNICODE_STRING altitudeUni = curCallback->CallbackEntry-
>Altitude;
                            RtlUnicodeStringToAnsiString(&altitudeAnsi, &altitudeUni, 1);

                            if (!strcmp(altitudeAnsi.Buffer, "363220")) { // Check if
this is BattlEye. If it is, disable the callback.
                                if (curCallback->PreOperation && oldCallbacks-
>PreOperationThread)
                                    curCallback->PreOperation =
(POB_PRE_OPERATION_CALLBACK)oldCallbacks->PreOperationThread;
                                if (curCallback->PostOperation && oldCallbacks-
>PostOperationThread)
                                    curCallback->PostOperation =
(POB_POST_OPERATION_CALLBACK)oldCallbacks->PostOperationThread;
                                RtlFreeAnsiString(&altitudeAnsi);
                                break;
                            }

                            RtlFreeAnsiString(&altitudeAnsi);
                        }

                        // Get the next callback.
                        curCallback = curCallback->CallbackList.Flink;
                    } while (curCallback != firstCallback);
                }
            }
        }
        __except (EXCEPTION_EXECUTE_HANDLER) {
            return;
```

```
        }
    }
}

NTSTATUS ioRecieved(PDEVICE_OBJECT pDeviceObject, PIRP IRP) {
    PIO_STACK_LOCATION pIoStackLocation = IoGetCurrentIrpStackLocation(IRP);
    size_t size = 0;

    // Handle the I/O request if we need to.
    if (pIoStackLocation->Parameters.DeviceIoControl.IoControlCode ==
REMOVE_BEOBJECT_CALLBACKS_IOCTL){
        OLD_CALLBACKS oldCallbacks = { 0 };
        DisableBEObjectCallbacks(&oldCallbacks);
        memcpy(IRP->AssociatedIrp.SystemBuffer, &oldCallbacks, sizeof(OLD_CALLBACKS));
        size = sizeof(OLD_CALLBACKS);
    }
    if (pIoStackLocation->Parameters.DeviceIoControl.IoControlCode ==
RESTORE_BEOBJECT_CALLBACKS_IOCTL) {
        RestoreBEObjectCallbacks((POLD_CALLBACKS)IRP->AssociatedIrp.SystemBuffer);
        size = 0;
    }

    // Finish off.
    IRP->IoStatus.Status = STATUS_SUCCESS;
    IRP->IoStatus.Information = size;
    IofCompleteRequest(IRP, IO_NO_INCREMENT);
    return(STATUS_SUCCESS);
}

NTSTATUS CatchCreate(PDRIVER_OBJECT pDriverObject) {
    return(STATUS_SUCCESS);
}

NTSTATUS CatchClose(PDRIVER_OBJECT pDriverObject) {
    return(STATUS_SUCCESS);
}

NTSTATUS DriverEntry(PDRIVER_OBJECT pDriverObject, PUNICODE_STRING pRegistryPath) {
    // Create the device and get everything set up.
    UNICODE_STRING deviceNameUnicodeString = { 0 }, deviceSymLinkUnicodeString = { 0 };
    RtlInitUnicodeString(&deviceNameUnicodeString, L"\\Device\\mmarkdrv");
    RtlInitUnicodeString(&deviceSymLinkUnicodeString, L"\\DosDevices\\mmarkdrv");
    IoCreateDevice(pDriverObject, 0, &deviceNameUnicodeString, FILE_DEVICE_KS,
FILE_DEVICE_SECURE_OPEN, 0, &deviceObj);
    IoCreateSymbolicLink(&deviceSymLinkUnicodeString, &deviceNameUnicodeString);

    // Get all the major functions set up.
    pDriverObject->MajorFunction[IRP_MJ_CREATE] = CatchCreate;
    pDriverObject->MajorFunction[IRP_MJ_CLOSE] = CatchClose;
    pDriverObject->MajorFunction[IRP_MJ_DEVICE_CONTROL] = ioRecieved;

    return(STATUS_SUCCESS);
}
```

LuluVisor BEDaisy Logs

Here is a sample of functions that the BE driver call, from iPower's logs from his hypervisor trace, which show you what BE is doing in Fortnite:

Code:

```
ExFreePool
IoQueryFileDosDeviceName
```

```
IoThreadToProcess
KeReleaseMutex
KeWaitForMutexObject
MmIsAddressValid
ObOpenObjectByName
ObReferenceObjectByHandle
ObfDereferenceObject
PsGetProcessInheritedFromUniqueProcessId
PsGetThreadProcessId
RtlCompareUnicodeString
RtlInitUnicodeString
ZwClose
ZwOpenFile
ZwQueryDirectoryObject
_wcsnicmp
```

Temporary Battleye Bypass for improperly implemented anticheat

It's happened a few times on a couple games where you can just unload the Battleye driver and the game doesn't stop running, it's easy to do, but unlikely it will work on most new games. @gulerardaeren posted this a while back, it has worked previously on Zula, Crossfire, Apex Legends and others as well

1. Open The Game
2. Open Process Hacker And Find BEService.exe
3. Right Click To BEService.exe and click suspend(this will require administator permission)
4. Open The Process Hunter and go to the Kernel Module
5. Find BEDaisy.sys , right click and click unload driver
6. If they didn't patched this method, You can even use cheat engine

Dumped Modules

The first thing you need to do to reverse engineer Battleeye is to dump the system driver and usermode modules from memory, you will find a couple pre-made dumps below. Once you have the dumps you can load them into IDA Pro and start looking around.

GH Battleye Bypass Resources

- iPower BE Logs & BEDaisy.sys Module Dump
- 02/07/2020 BEDaisy.sys dump
- Old Battleye Bypass Source Code
- Release - BattlEye Bypass [+Tested on Rainbow Six Siege] (Driver) (Source)
- c5's old IDA Scripts for Analyzing BE

External Resources

The most important thing you can do to learn about BattleEye is to watch this video made by the DayZ developers about how Battleeye helps them stop cheaters. This is also an excellent video for anyone wanting to learn about anticheat.

https://www.youtube.com/watch?v=0M0xBMEuWdU

- Back Engineering - Badeye
- vmcall/battleye_emulation - battle eye usermode emulator
- vmcall/eye_mapper - BattlEye x64 usermode injector (patched)

- ArcherPeng/FuckBattlEye - Anticheat Bypass exploiting leaked handles.
- Tai7sy/BE_Fuck - BattlEye Emulator
- gigabitwize/dank - windows 8.1 battleeye bypass
- huoji120-battleye random battle eye code
- daswareinfach/Battleye-VAC-EAC-Kernel-Bypass
- ContionMig/LSASS-Usermode-Bypass
- Schnocker/HLeaker
- yunseok/HWIDbypass
- Neijwiert/Feeder
- Schnocker/NoEye

Credits: @iPower, @_xeroxz, douggem, vmcall & everyone at secret.club

BattlEye client emulation - Bottleye

The popular anti-cheat BattlEye is widely used by modern online games such as Escape from Tarkov and is considered an industry standard anti-cheat by many. In this article I will demonstrate a method I have been utilizing for the past year, which enables you to play any BattlEye-protected game online without even having to install BattlEye.

BattlEye initialisation

BattlEye is dynamically loaded by the respective game on startup to initialize the software service ("BEService") and kernel driver ("BEDaisy"). These two components are critical in ensuring the integrity of the game, but the most critical component by far is the usermode library ("BEClient") that the game interacts with directly. This module exports two functions: GetVer and more importantly Init.

The *Init* routine is what the game will call, but this functionality has never been documented before, as people mostly focus on BEDaisy or their shellcode. Most important routines in *BEClient*, including *Init*, are protected and virtualised by VMProtect, which we are able to devirtualise and reverse engineer thanks to vtil by secret club member Can Boluk, but the inner workings of *BEClient* is a topic for a later part of this series, so here is a quick summary.

https://www.youtube.com/watch?v=GeZz8xtdLgQ

Bypassing BattlEye from user-mode

Today we'll talk about how BattlEye does integrity checks for loaded images, as well as implementing a work-around for these checks.

Image integrity checks

BattlEye does checks on images that get loaded by opening a handle to the file on disk with CreateFile, after this handle's open, it retrieves certificate details for the file, and checks if it's one of the blacklisted certificates. If it is, the file gets blocked from loading and BattlEye notifies you that a blacklisted file was attempting load.

Continue reading @ secret.club - Bypassing BattlEye from user-mode

BattlEye reverse engineer tracking

Preface

Modern commercial anti-cheats are faced by an increasing competetiveness in professional game-hack production, and thus have begun implementing questionable methods to prevent this. In this article, we will present a previously unknown anti-cheat module, pushed to a small fraction of the player base by the commercial anti-cheat BattlEye. The

prevalent theory is that this module is specifically targeted against reverse engineers, to monitor the production of video game hacking tools, due to the fact that this is dynamically pushed.

Shellcode ??
The code snippets in this article are beautified decompilations of shellcode that we've dumped and deobfuscated from BattlEye...

Continue reading @ [BattlEye reverse engineer tracking - secret.club]

BattlEye anticheat: analysis and mitigation

BattlEye is a prevalent german third-party anti-cheat primarily developed by the 32-year-old founder *Bastian Heiko Suter*. It provides game publishers easy-to-use anti-cheat solutions, using generic protection mechanisms and game-specific detections to provide optimal security, or at least tries to. As their website states, they are always staying on top of state-of-the-art technologies and utilizing innovative methods of protection and detection, evidently due to their nationality: QUALITY MADE IN GERMANY. BattlEye consists of multiple organs that work together to catch and prevent cheaters in the respective games that pay them. The four main entities are:

- **BEService** - Windows system service that communicates with the BattlEye server *BEServer*, which provides *BEDaisy* and *BEClient* server-client-communication capabilities.
- **BEDaisy** - Windows kernel driver that registers preventive callbacks and minifilters to prevent cheaters from modifying the game illicitly.
- **BEClient** - Windows dynamic link library that is responsible for most of the detection vectors, including the ones in this article. It is mapped into the game process after initialization.
- **BEServer** - Proprietary backend-server that is responsible for collecting information and taking concrete actions against cheaters.

Shellcode
Recently, a dump of BattlEye's shellcode surfaced on the internet, and we decided to make a write-up of what exactly the current iteration of BattlEye is actively looking for. We have not worked on BattlEye for the past 6 months, so the last piece of shellcode we have dumped is most likely obsolete. Miscellaneous parts of code were recognized completely from memory in this recent dump, suggesting that BattlEye only appends to the shellcode and does not remove previous detection procedures.

BattlEye shellcode updates

Anticheats change as time goes on, features come and go to maximize the efficiency of the product. I did a complete write-up of BattlEye's shellcode a year ago on my blog, and this article will merely reflect the changes that have been made to said shellcode.

Blacklisted Timestamps

Last time I analyzed BattlEye, there were only two compile-time datestamps in the shadowban ban list, and it seems like they've decided to add a lot more:

- 0x5B12C900 (action_x64.dll)
- 0x5A180C35 (TerSafe.dll, Epic Games)
- 0xFC9B9325 (?)
- 0x456CED13 (d3dx9_32.dll)
- 0x46495AD9 (d3dx9_34.dll)
- 0x47CDEE2B (d3dx9_32.dll)
- 0x469FF22E (d3dx9_35.dll)

- 0x48EC3AD7 (D3DCompiler_40.dll)
- 0x5A8E6020 (?)
- 0x55C85371 (d3dx9_32.dll)
- 0x456CED13 (?)
- 0x46495AD9 (D3DCompiler_40.dll)
- 0x47CDEE2B (D3DX9_37.dll)
- 0x469FF22E (?)
- 0x48EC3AD7 (?)
- 0xFC9B9325 (?)
- 0x5A8E6020 (?)
- 0x55C85371 (?)

I've failed to identify the rest of the timestamps, and the two 0xF******* are hashes produced by visual studio reproducible builds. If anyone can identify the timestamps, please hit me up on twitter ?

Thanks to @mottikraus and T0B1 for identifying some of the timestamps.

continue reading @ secret.club...

Battleye Stack Walking

With game-hacking being a continuous cat and mouse game, rumours about new techniques spread like fire. As such in this blog post we will take a look into one of the new heuristic techniques that BattlEye, a large anti-cheat provider, has recently added to its arsenal. Most widely known as stack walking This is usually done by hooking a function and traversing the stack to find out who exactly is calling said function. Why would one do this? Just like any other program, video game hacks have a set of well known functions that they utilize to get keyboard information, print to the console or calculate certain mathematical expressions. Video game hacks also like to attempt to hide their existence, be it in memory or on disk, so that the anti-cheat software does not find it. What these cheat programs forget is that they regularly call functions in other libraries, and this can be exploited to heuristically detect unknown cheats. By

implementing a stack walking engine on prevalent functions like std: rint, you will be able to find these cheats even if they disguise themselves.

BattlEye **has** implemented "stack walking", even though this has not been publicly proved and prior to this article was just rumors. Note the quotes around stack walking, because what you will see here is not true stack walking, this is merely a return address check and a caller dump combined. A true stack walker would traverse the stack and generate a proper callstack.

If you want to learn how to bypass BattlEye anticheat, these Secret Club articles should give you a great starting place.

Battleye Hypervisor Detection

The cat and mouse game of game-hacking continues to fuel the innovation of exploitation and mitigation. The usage of virtualization technology in game-hacking has exploded ever since copy-pastable hypervisors such as Satoshi Tanda's DdiMon and Petr Beneš' hvpp hit the scene. These two projects are being used by most of the paid cheats in the underground hacking scene, due to their low barrier of entry and extensive documentation. These releases have with high certainty sped up the hypervisor arms race that is now beginning to show its face in the gamehacking community. Here's what the administrator at one of the worlds largest game-hacking communities, wlan, says about the situation:

With the advent of ready-made hypervisor solutions for game hacking it's become unavoidable for anti-cheats such as BattlEye to focus on generic virtualization detections

The reason hypervisors are so wide-spread now is because of recent developments in kernel anti-cheats leaving very little room for hackers to modify games through traditional means. The popularity of hypervisors could be explained by the simplicity of evasion, since virtualization enables you to more easily hide information from the anti-cheat, through mechanisms such as syscall hooks and MMU virtualization.

BattlEye has recently implemented a detection of generic hypervisors such as the previously mentioned platforms (DdiMon, hvpp) using time-based detection. This detection aims to spot abnormal time values in the instruction CPUID. CPUID is a relatively cheap instruction on real hardware, and will generally only require two hundred cycles, where as in a virtualized environment it may take up to ten times as long due to the overhead incurred by an introspective engine. An introspective engine is not like any real hardware which just performs the operation as is expected, it monitors and conditionally changes the data returned to the guest based on arbitrary criteria.

Fun fact: CPUID is commonly used in these time based detection routines because it is an unconditionally exiting instruction as well as an unprivileged serializing instruction. This means that CPUID acts as a 'fence' and ensures that instructions before or after it are completed and makes the timing independent of typical instructions reordering. One could use instructions like XSETBV which also unconditionally exits, but to ensure independent timing would need to use some sort of FENCE instruction so that no reordering occurs before or after that would affect the timings reliability.

hypervisor-detection.html']continue reading @ secret.club...[/URL]

BattlEye communication hook

To combat masses of video game hackers, anti cheat systems need to collect and process a lot of information from clients. This is usually usually done by sending everything to the servers for further analysis, which allows the attackers to circumvent these systems through interesting means, one of them being hijack of the communication routine.

If an anti cheat is trying to detect a certain cheat by, for example, the name of the process that hosts the cheat code, it will usually parse the entire process list and send it to the server. This way of outsourcing the processing prevents cheaters from reverse engineering the blacklisted process names, as all they can see is that the *entire* process list is sent to the anti cheat server. This is actually becoming more and more prevalent in the anti cheat community, which raises some serious privacy concerns, simply due to the sheer amount of information being sent to a foreign server.

BattlEye, one of the world's most installed anti cheats, uses such a routine to send data to their master server over UDP. This function is usually referred to as battleye::send or battleye::report (as in my previous articles). It takes two parameters: buffer and size. Every single piece of information sent to the BattlEye servers is passed through this function, making it very lucrative for hackers to intercept, possibly circumventing every single protection as the game can't report the anomalies if a hacker is the middleman of communcations. Few cheat developers are actively using this method, as most of them lack the technical skills to reverse engineer and deobfuscate the dynamically streamed modules that BattlEye heavily relies on, but in this post i will shed some light on how this communication routine is being actively exploited, and how BattlEye has tried to mitigate it

continue reading @ secret.club...

How Escape from Tarkov ensures game integrity

Game-hacking is an always-changing landscape, and this requires anti-cheat developers to innovate and implement unique, unidentified detection mechanisms. In this article I will shed some light on the mysterious routines that are getting hundreds of cheaters banned in Escape from Tarkov. So let's start from the beginning.

Escape from Tarkov (herein "Tarkov") runs on the game engine Unity through Mono, which opens up for some interesting security issues that game-hackers can abuse to gain an advantage while playing. First of all, the Unity game assemblies are *very* hard to integry-check when they've been JIT-compiled. This is because you can't simply store a hash value1 of the code, as the JIt-compiled methods might differ depending on what processor features are enabled.

This leaves the anti-cheat developers in a tough spot. It is not possible to ensure the integrity of JIT-compiler functions without either:

- Initialising before the game does then hooking the responsible JIT-engine. This hook can be used to cache hashes for all compiled functions for later comparison
- Resorting to alternative ways for ensuring game integrity, like checking image metadata.

BattlEye..?

While Tarkov actually has integrity checks (simple file hashing) in their Battlestate Games launcher application, this is trivial to patch out of the executable by opening the launcher executable in a tool like dnSpy and simply removing the entire thing. The fact that this integrity check (internally called "consistency check" in the launcher) was so easy to circumvent, enabled thousands of cheaters to simply patch the game assembly on disk. This could include features such as "wallhack", "no recoil" et cetera.

It seems like Battlestate Games got tired of this vulnerability, and to fix it, they *likely* called up the developers of the commercial anti-cheat BattlEye, which they've been utilizing for quite some time now. This article will explore a previously-unknown anti-cheat module that is being dynamically streamed and executed to Tarkov players circa 15-20 minutes into their raids.

continue reading @ secret.club...

Cracking BattlEye packet encryption - Escape From Tarkov

Recently, Battlestate Games, the developers of Escape From Tarkov, hired BattlEye to implement encryption on networked packets so that cheaters can't capture these packets, parse them and use them for their advantage in the form of radar cheats, or otherwise. Today we'll go into detail about how we broke their encryption in a few hours.

Analysis of EFT

We started first by analyzing Escape From Tarkov itself. The game uses Unity Engine, which uses C#, an intermediate langauge, which means you can very easily view the source code behind the game by opening it in tools like ILDasm or dnSpy. Our tool of choice for this analysis was dnSpy.
Unity Engine, if not under the IL2CPP option, generates game files and places them under GAME_NAME_Data\Managed, in this case it's EscapeFromTarkov_Data\Managed. This folder contains all the dependencies that the engine uses, including the file that contains the game's code which is Assembly-CSharp.dll, we loaded this file in dnSpy then searched for the string encryption, which landed us here:

```
channelCombined.bool_2 = encryptionEnabled;
channelCombined.bool_3 = decryptionEnabled;
channelCombined.bool_2 = false;
logger.LogInfo("{0}:{1}, {2}:{3}", new object[]
{
    "_encryptionEnabled",
    channelCombined.bool_2,
    "_decryptionEnabled",
    channelCombined.bool_3
});
```

How to bypass XignCode Anticheat Guide - XignCode3

This is compilation of all information and resources regarding XignCode Anticheat, this should help anyone get a jump start on making a bypass for this anticheat.

What is XignCode Anticheat?

Xigncode anticheat is a software developed by Wellbia that is designed to prevent cheating in online games. It is designed to detect and prevent the use of unauthorized third-party programs that modify game data or give unfair advantages to players. Xigncode anticheat operates by scanning a player's computer for any suspicious software or files and compares them with a list of known cheats.

If a cheat is detected, Xigncode anticheat can block the user's access to the game, and in some cases, it can even ban the player from the game permanently. While Xigncode anticheat has been criticized by some players for its invasive scanning practices, it remains a popular choice for many game developers due to its effectiveness in preventing cheating.

Xigncode Versions

Xigncode mostly protect Asian games, many of which have small player bases and cannot afford new iterations of the anticheat. Xigncode has been around for a long time and has many different versions. The latest version that newer games use is XignCode3.

Xigncode is the older version and was widely used in online games until Xigncode3 was released. Xigncode 3, on the other hand, is a heavily upgraded version of the anti-cheat software. It is designed to be more efficient and effective in detecting cheating activities in online games. It also has better compatibility with newer versions of Windows and other operating systems.

Keep in mind that the game developers can decide which anticheat features to use in their game.

Basically there are 3 levels of Xigncode implementation:

- Basic Xigncode game process protection
- Custom, game specific protections
- Heartbeat

Some games you just need to disable the basic Xigncode memory protections and then you can inject.

Before Xigncode 3, it mostly prevented debuggers, code injection and other basic process and memory manipulations of the main game process. Xigncode3 is much more invasive, scanning all your processes and sending additional information to their servers.

Overall, Xigncode3 is considered to be more advanced and effective in preventing cheating in online games compared to its predecessor. However, both versions are still used in many online games today.

Old versions of Xigncode are relatively easy to bypass if you are an expert game hacker.

Is Xigncode better than Battleye or EAC?

The old version is not very good, but because it's a kernel anticheat it does prevent the majority of cheaters. Xigncode3 as deployed on it's flagship games, on the other hand is about 75% as effective at Battleye and 50% as effective as EAC.

If you can bypass EAC or Battleye, you should have no problem bypassing Xigncode3.

Games that use Xigncode3

You can get a full list of games which use Xigncode3 here.
These are the most popular games which have used Xigncode at some point:

- Aion
- Alliance of Valiant Arms
- ArcheAge
- Atlantica Online
- Black Desert Online
- Blade and Soul
- Cabal 2
- Combat Arms
- CrossFire
- Dekaron
- Echo of Soul
- Elsword
- Final Fantasy XIV
- Heroes & Generals
- KurtzPel
- Lost Ark
- MapleStory 2
- Moonlight Blade
- PUBG
- Paladins
- Phantasy Star Online 2
- Point Blank
- Ragnarok Online
- Riders of Icarus
- Special Forces 2
- Sudden Attack 2
- TERA
- Tera
- Vindictus
- Warface
- Warframe
- Wolfteam
- Zula

How to Bypass XignCode?

If you haven't been hacking for at least a year, you have no business trying to bypass an anticheat. Just stop and focus on learning hacking. If you're new to anticheat, read our general guide: How to Get Started with AntiCheat Bypass

There is no bypass that you just copy and paste to hack these games. The information and resources we're providing here are mostly outdated but will aid you in reversing the anticheat.

Xigncode3 can be very difficult to bypass in some games depending on the implementation, so keep your goals aligned with your experience and skill level.

What is a heartbeat in terms of anticheat?

Heartbeat is a technique used to detect tempering of the xigncode anticheat. The anticheat is in constant communication with the server, the communication is heavily obfuscated, abstracted and the communication is verified at many different places. Any tampering with the anticheat will cause the server to disconnect the client.

If the game has heartbeat sometimes you can disable the anticheat just long enough to dump the modules from memory so you can reverse engineer them or just long enough to find a pointer. Most people say you get disconnected with 30-120 seconds.

Logged XignCode Files

These are files we have seen when reverse engineering games.

- **x3.xem** - the main xigncode3 DLL
- **xhunter1.sys** - xigncode kernel mode driver
- **xm.exe**
- **xmag.xem**
- **xsg.xem**
- **xxd.xem**

What does Xigncode3 it detect?

It can detect anything that happens in usermode, but only some of the things that happen in kernel land. If you want to bypass it easily, use a manually mapped kernel driver to interact with the game process.

Detections

- All debuggers including Cheat Engine, x64dbg etc...
- All patches of code pages, it uses CRC memory integrity checks,
- DirectX and other common hooks
- Process suspending
- Anything wierd in DLL main
- Thread creation
- Common Windows APIs are hooked
- Virtual function hooks are detected
- Manual mapping works on some old versions
- Removing the PE Header used to make dll's to become undetected
- Removing the xhunter1 service used to prevent future detections and dll injection detections
- Hook to NtQueryInformationProcess, NtQueryVirtualMemory, NtReadVirtualMemory, NtQueryInformationThread,
- NtOpenFile, NtWow64QueryInformationProcess64, NtWow64QueryVirtualMemory64, NtWow64ReadVirtualMemory64 to view anything involving to your dll and xigncode
- Detects LoadLibrary injection, CreateThread, GetAsyncKeyState, CreateFont, LdrLoadDll, LoadLibraryA, LoadLibraryW, LoadLibraryExA, LoadLibraryExW, GetModuleFileName.
- Always obfuscate / encrypt your dll.
- They register an callback on the object manager by using ObRegisterCallback().
- That means that after the rootkit is enabled xign is able to trace all access you make to the games process.
- Checks each module's crc / md5 with a internal list.

- CreateRemoteThread can be used.
- Xign checks the stack frame from NtUserGetAsyncKeyState
- Spoof return addresses after looking into SetWindowsHookEx and GetWindowLongPtr
- Use low level keyboard / mouse hooks
- for d3d9 use a vtable

GetAsyncKeyState Detection

Because GAKS is very commonly used in hacks, they very easily detect you using it or hooking it. To bypass this detection use a different method of reading keyboard keys or as @Broihon has said, wait for Xigncode to hook it, then hook it afterwords, something similar to:

C++:

```
BYTE * pGAKS = reinterpret_cast<BYTE*>(GetAsyncKeyState);
BYTE Orig[10];
memcpy(Orig, pGAKS, 10);

bool bChanged = false;

while (!bChanged)
{
    for (UINT i = 0; i != 10; ++i)
        if (pGAKS[i] != Orig[i])
            bChanged = true;
    Sleep(100);
}

DWORD dwOld = 0;
VirtualProtect(pGAKS, 10, PAGE_EXECUTE_READWRITE, &dwOld);
memcpy(pGAKS, Orig, 10);
VirtualProtect(pGAKS, 10, dwOld, &dwOld);
```

How to uninstall Xigncode?

Xigncode is a rootkit and therefore has complete access to your entire computer and can examine any file or process on it. It only runs when you're playing games, but it's still unnerving. If you want to remove it, it's important to ensure that any games or programs that utilize Xigncode are closed.

1. Open the Control Panel
2. Navigate to Programs and Features
3. Look for Xigncode in the list
4. Click "Uninstall" and follow the prompts
5. Restart your computer

If it's still not removed, use these commands in an elevated command prompt:
Code:

```
net stop xhunter1
reg delete HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\xhunter1
del C:\Windows\xhunter1.sys
```

Reversing Engineering Xigncode3 Series

- Reversing XignCode3 Driver – Part 1 – Identifying the Driver Entry Point
- Reversing XignCode3 Driver – Part 2 – Analyzing init functions
- Reversing XignCode3 Driver – Part 3 – Analyzing dispatch functions
- Reversing XignCode3 Driver – Part 4.1 – Registering Notify and Callback Routines

Newest Bypass XignCode3

They year is 2023, if you want the most up to date sources available, check these:

- VirtualPuppet/XignCode3-bypass-alternative
- VirtualPuppet/XignCode3-bypass

Old Source Codes

The rest of these source codes will be for the old versions of Xigncode, but may be helpful for your research

C++:

```
PBYTE FindStartOfFunc(PBYTE Addy)
{
    if (!Addy) return Addy;
    while (true) if (compare((PBYTE)"\x55\x8B\xEC", "xxx", Addy--)) return ++Addy;
}

PBYTE FindPush(PBYTE sig, PCHAR mask, DWORD dwBase, DWORD dwLen)
{
    if (!dwBase) return nullptr;
    BYTE PushSig[5] = { 0x68, 0, 0, 0, 0 };
    *(PDWORD)(&PushSig[1]) = FindSignature(sig, mask, dwBase, dwLen, 0);
    if (*(PDWORD)(&PushSig[1]) == NULL) return NULL;
    return (PBYTE)FindSignature(PushSig, "xxxxx", dwBase, dwLen, 0);
}

bool bTriggered = false, bSuccess = false;
void bypass()
{
    DWORD dwCShell = FindCShell();
    if (dwCShell != NULL)
    {
        PBYTE BypassSig = FindPush((PBYTE)"XIGNCODE", "xxxxxxxxx", dwCShell, 5000000);
        if (BypassSig != nullptr)
        {
            PBYTE BypassFunc = FindStartOfFunc(BypassSig);
            if (BypassFunc && !memcmp(BypassFunc, (PBYTE)"\x55\x8B\xEC", 3))
            {
                Wrt((PBYTE)BypassFunc, (PBYTE)"\xB0\x01\xC3", 3);
                bSuccess = true;
            }
        }
    }
    bTriggered = true;
}
cBreakpoint* bp = NULL;
PBYTE pcheck = 0;
LONG WINAPI ExceptionHandler(EXCEPTION_POINTERS* e)
{
```

```
    if (e->ExceptionRecord->ExceptionCode != EXCEPTION_SINGLE_STEP) return
EXCEPTION_CONTINUE_SEARCH;
    if (e->ContextRecord->Eip == (DWORD)pcheck)
    {
        e->ContextRecord->Esp -= 4;
        *(PDWORD)(e->ContextRecord->Esp) = e->ContextRecord->Eip + 0x2;
        e->ContextRecord->Eip = e->ContextRecord->Edx;
        bypass();
        return EXCEPTION_CONTINUE_EXECUTION;
    }
    return EXCEPTION_CONTINUE_SEARCH;
}

void Start()
{
    Sleep(1000);
    AntiHWIDBan();
    while (pcheck == nullptr)
    {
        Sleep(30);
        pcheck = FindPush((PBYTE)"DIRECTSHOW\x00", "xxxxxxxxxx",
(DWORD)GetModuleHandleA("wolfteam.bin"), 5000000);
    }
    pcheck -= 2;
    bp = new cBreakpoint(ExceptionHandler);
    bp->SetBP((DWORD)pcheck);
    while (!bTriggered) Sleep(1000);
    delete bp;
}
```

Instructions

1. Go into the XIGNCODE root folder.
2. Rename “x3.xem” to “x3.dummy”.
3. Enter the code you want under the DllMain.
4. Compile the DLL with the code provided down bellow under the name “x3.dll”.
5. Rename “x3.dll” to “x3.xem” and put it into the XIGNCODE root folder.
6. Start your game and the code should be executed and not be detected.

C++:

```
typedef int32_t(__stdcall *t_x3_Dispatch)(OUT void *Function,
                                          IN uint32_t Type);

static t_x3_Dispatch o_x3_Dispatch = nullptr;

void __stdcall DllMain() {
  MessageBoxA(0, "XIGNCODE3 ded", "kek", 0);

  // You put your code here
}

__declspec(dllexport) int32_t
    __stdcall x3_1(void *FunctionAddress, uint32_t Type) {

  if (o_x3_Dispatch == nullptr) {
    std::string ModulePath;
    ModulePath.resize(MAX_PATH);
```

```
    if (!GetModuleFileNameA(NULL, const_cast(ModulePath.data()), MAX_PATH)) {
      MessageBoxA(0, "GetModuleFileNameA failed!", "Error", 0);
      return 80000000;
    }

    std::string xignf = ModulePath.substr(0, ModulePath.find_last_of(""));
    xignf += "\\XIGNCODE\\x3.dummy";
    HMODULE hX3 = LoadLibraryW(xignf.c_str());

    if (hX3 == nullptr) {
      MessageBoxA(0, "LoadLibraryA failed!", "Error", 0);
      return 80000000;
    }

    o_x3_Dispatch = reinterpret_cast(GetProcAddress(hX3, reinterpret_cast(1)));

    if (o_x3_Dispatch == nullptr) {
      MessageBoxA(0, "GetProcAddress failed!", "Error", 0);
      return X3_NOT_INITIALIZED;
    }
    DllMain();
  }
  return o_x3_Dispatch(FunctionAddress, Type);
}
```

I found this on pastebin

- XC has a single-call which starts the anti-cheat (it loads x3.xem) just nop that call and fix some of the jumps in that region and you should get a bypass until heartbeat

Hook the following and filter out anything related to your DLL:

- NtQueryInformationProcess
- NtQueryVirtualMemory
- NtReadVirtualMemory
- NtQueryInformationThread
- NtOpenFile
- NtWow64QueryInformationProcess64
- NtWow64QueryVirtualMemory64
- NtWow64ReadVirtualMemory64

Xigncode Bypass Resources

- Xigncode Bypass - Application Shim Attack
- xingcode3 - x3 and xcorona unpacked files
- Bypass XignCode GetaSyncKeyState Detection
- Xigncode game differences
- Is There Any Way? to bypass xigncode3 heartbeat
- xigncode.
- Xigncode 3 bypass for Wolfteam

How to Bypass Xigncode3

The most important fact about Xigncode is that it heavily depends on the game. All those features Rake mentioned can be implemented sepereately. Some games have all of the mechanics implemented, other games only the most basic window detection.

I messed with a Xigncode protected game which only did the following:
- heartbeat
- close handles to the game ONCE (you CAN reopen it after that)
- window detection which ONLY minimizes stuff like Cheat Engine but DOESN'T close the game
Other version are in fact much harder to bypass.

XC's module detection fully relies on NT- and WINAPIs. So you could technically hook all modules listing related APIs but that's a lot of work and you'd also hook their functions which check for hooks. So just stick to manual mapping. They don't have any pattern/signature based module detection. They use NtQueryVirtualMemory which uses kernel information. That's why any usermode cloaking of your dll is useless.
XC calls some native functions directly by using syscall (x64) or the wow64 transition thingy (x86) which makes even finding what native functions they use even more annoying.

As Rake also said using GetAsyncKeyState is detected since XC hooks it and checks the callstack. Unhooking works fine but will probably be detected at some point which is why one should probably switch to lowlevel keyboard hooks now.
Edit: On some systems it might hook NtUserGetAsyncKeystate instead or both. Same method applies, if you want to learn how to bypass Xigncode3.
When it comes to DirectX hooking there are a few reliable methods but the one I prefer is the following. XC doesn't have code checksum checks for the game's code. Or at least I've never encountered that. They hash and check most windows modules but not the game's module. This means we can simply hook into the game's code when it calls the various DX functions (doesn't matter what version). Leave the D3DX.dll alone and just hook into the game directly and you'll be fine. You can also hook into the game's vtable. That undetected aswell.
As for thread creation I've never had trouble spawning as many threads as I liked. But this of course can change or maybe has already been changed.
In case the game properly protects the game's process from being opened you can always hijack a handle. Inheriting an existing handle to a child process which then eg. injects a dll worked fine for me. Again - could've been changed by now but I doubt that. In case it has been changed you can always write more shellcode to the owner of the original handle and inject from there without inheriting the handle to another process.

@IXSO sent me some info that he had regarding a game that had a pretty poor implementation the anticheat. If you want to bypass Xigncode3 check this list of hooked functions below.

The version I have to work with is super buggy (same as game) and not the latest. The developers copied the Xigncode folder from ava and did a terrible job implementing it, therefore, I can call any API (GAKS, LL, CT, etc.) without detection. Idk how much help I will be as all I've done is hooked a couple WINAPIs so that I can use whatever I want except for ollydbg (I believe Themed is the one detecting it tho)

The APIs I'm hooking

- user32.PostMessageW - used to minimize CE, sends WM_SYSCOMMAND + SC_MINIMIZE in a loop
- Advapi32.OpenSCManager - removes XC3 service
- kernel32.DuplicateHandle - prevents XC3 from closing handles. Gets called with 1st 2 parameters being -1, which I believe stands for "All"
- kernel32.ExitProcess - never called, still hooking it
- kernel32.TerminateProcess - same
- kernel32.TerminateThread - same
- kernel32.IsDebuggerPresent - idk if it's called, but it's still better to hook it
- ntdll.ZwTerminateProcess - hook it just in case
- ntdll.ZwSetInformationThread - iirc I hooked this hoping to avoid Themida debugger detections, but I'm not sure

Hackshield Anticheat Bypass Information

Here's all our information related to making a Hackshield bypass, in one thread.

Can else can share some info on HackShield? Specifically what it's like in the past few years...Because some of this is pretty outdated.

Before you focus on Hackshield you must understand the basics of Anticheat Guide - How to Get Started with AntiCheat Bypass

Here's what I dug up so far (keep in mind lots of this info is old, but is useful for research):

Hackshield has been around since 2005, made by AhnLab Inc. It's used on lots of MMOs, many Nexon and NCSoft games. Interestingly enough AhnLab makes antivirus software also. It is a kernel mode anticheat but has been bypassed many times in the past, meaning it must not be too difficult.

Hackshield source code

The Hackshield source code was leaked around 2010, if you are real good at DuckDuckGoing you can find it, very interesting read but not too useful as it's probably got a lot of updates, but it's the only anticheat source code I know about. Thank you @timb3r for mentioning it. I found it online @ HackShieldR.5.6.6.1(build235).zip

Detections:
Uses signature detection to detect hacking programs such as cheat engine, injectors etc...
There is a kind of heartbeat, where the server and client talk continuously and monitor for changes
Blocks hooking and sending messages to window
Blocks debuggers

Games that use It:
Combat Arms
PointBlank
??

There is an excellent writeup on the older version @ HackShield Analysis - Anti-Cheat Systems - Games Research Community

Here's a relatively new bypass on GitHub that is very nice

VirtualPuppet/HackShield-bypass

Recent Bypasses from AIRRIDE for v5.6.34.449, v5.7.6.502 & v5.7.20.616

Hackshield Anticheat Bypass Information

First bypass I found from a few years ago:

C++:

```
__declspec (naked) void HS_PATCH_1()
{
    __asm {
        inc     eax
        add[esi + ecx - 7Fh], bh
        inc     byte ptr[eax]
        add[eax + 3067D00h], dl
```

```
        xor     eax, dword_1002FD44
        push     36h
        lea     edi, [ebp - 122Ch]
        retn
    }
}
__declspec (naked) void HS_PATCH_2()
{
    __asm {
        inc     eax
        add[esi + ecx - 7Fh], bh
        inc     byte ptr[eax]
        add[eax + 3067D00h], dl
        xor     eax, dword_1002FD44
        mov     eax, ecx
        mov     edx, ecx
        add     eax, esi
        retn
    }
}

__declspec (naked) void sub_hs_detect_sumthin()
{
    char time;
    time = Get_Time(2);//format = 2  ( "[%H:%M:%S]" )
    AddLog("%s - HackShield Detect Something...\n", time);
}

//sub to get ehsvc handle
int Get_Handle()
{
    int result;
    result = GetModuleHandleA("EhSvc.dll");
    EhSvc = result;
    return result;
}

void Detour_Hs()
{
    Sleep(1000);
    char time = Get_Time(2); //, 2 = "[%H:%M:%S]"
    AddLog("%s - Detouring HackShield->", current_time);
    int v2 = sub_1001883C(0x900000);
    sub_10016A80(v2, 0x401000, 0x900000);//bit complicated
    dword_1002FD44 -= 0x401000;

    while (!EhSvc)
    {
        EhSvc = GetModuleHandleA("EhSvc.dll");
        Sleep(100);
    }
    //
    DWORD HS1 = FindPattern(EhSvc, 0x90000,
(PBYTE)"\x74\x06\x83\x7D\x0C\x00\x75\x0F\x6A\x57", "xxxxxxxxxx");
    DWORD HS2 = FindPattern(EhSvc, 0x90000,
(PBYTE)"\x8D\xBD\xD4\xED\xFF\xFF\xF3\xA5\x8B\x53\x0A\x89\x95\xD0\xED\xFF\xFF\x33\xC0\x66\
x8B\x43\x08", "xxxxxxxxxxxxxxxxxxxxxxx");
    DWORD HS3 = FindPattern(EhSvc, 0x90000,
(PBYTE)"\x74\x09\xC7\x45\xFC\x00\xEB\x07\xC7\x45\xFC", "xxxxxxxxxxxxxx");
    DWORD HS4 = FindPattern(EhSvc, 0x90000, (PBYTE)"
\x8B\xC1\x8B\xD1\x03\xC6\x3B\xFE\x76\x08", "xxxxxxxxxx");
    DWORD HS5 = FindPattern(HS4 + 0x0A, 0x40000, (PBYTE)"
\x8B\xC1\x8B\xD1\x03\xC6\x3B\xFE\x76\x08", "xxxxxxxxxx");
```

```
    DWORD dword_1002BF4C = (0x74);
    DWORD dword_1002BF50 = (0x8D, 0xBD, 0xD4, 0xED, 0xFF, 0xFF);
    DWORD dword_1002BF58 = (0x74);
    DWORD dword_1002BF5C = (0x8B, 0xC1, 0x8B, 0xD1, 0x03, 0xC6);

    if (compare(HS1, &dword_1002BF4C, 1) //compare DWORD1,DWORD2,lenght
        || compare(HS2, dword_1002BF50, 6)
        || compare(HS3, &dword_1002BF58, 1)
        || compare(HS5, dword_1002BF5C, 6))
    {
        AddLog("Error, HackShield module changed!");
        sub_10017150(1);//this is callind sub that is calling another sub that kill
warrock
    }

    DWORD bit_1 = (0xEB); // (JMP SHORT)
    DWORD bit_2 = (0xE8, 0x00, 0x90);//call  something
    DWORD bit_3 = (0xE9, 0x00, 0x90);//jmp somewhere
    DWORD bit_4 = (0x4F, 0x4B, 0x21, 0x0A);

    sub_1000C4C8((PBYTE)HS1, 0x90, 1);
    sub_1000C514((PBYTE)HS2, (PBYTE)HS_Patch_1, 0xEB, 1);
    sub_1000C4C8((PBYTE)HS3, (PBYTE)bit_1, 1);
    sub_1000C514((PBYTE)HS5, (PBYTE)HS_Patch_2, bit_2, 6);
    sub_1000C514((PBYTE)0x681240, (PBYTE)sub_hs_detect_sumthin, bit_3, 6);
    AddLog((const char *)bit_4);//hmm confusing

    time1 = Get_Time(2);//format = 2  ( "[%H:%M:%S]" )
    AddLog("%s - Checking Dll->", time1);

    int check = sub_10017024(10);//compare if (10 > 0xFFFFFFE0 )return 0;

    if (check)
        v12 = sub_10010960();
    else
        v12 = 0;
    sub_10010BCF(dword_1002FBDC);
    AddLog("OK!\n");
}
```

Another bypass from a few years ago Author: Mafia67

C++:

```
BOOL WriteMemory (VOID *lpMem, VOID *lpSrc, DWORD len)
{
  DWORD lpflOldProtect, flNewProtect = PAGE_READWRITE;
  unsigned char *pDst = (unsigned char *)lpMem,
  *pSrc = (unsigned char *)lpSrc;
  if (VirtualProtect(lpMem,len,flNewProtect,&lpflOldProtect))
  {
              while(len-- >0) *pDst++ = *pSrc++;
              VirtualProtect(lpMem,len, lpflOldProtect,&lpflOldProtect);
              FlushInstructionCache(GetCurrentProcess(), lpMem, len);
              return 1;
  }
  return 0;
}

void HSBypass (void)
{
    DWORD dwEHSVC = 0;
```

```
    do
    {
        dwEHSVC = (DWORD)GetModuleHandle("EhSvc.dll");
        Sleep(250);
    }while(!dwEHSVC);

        WriteMemory((LPVOID)(dwEHSVC + 0x003D67F), (LPVOID)"\x03\xD2", 2);
        WriteMemory((LPVOID)(dwEHSVC + 0x003F77D), (LPVOID)"\xB8\x00\x00\x00\x00", 5);
        WriteMemory((LPVOID)(dwEHSVC + 0x000A1A0), (LPVOID)"\xC2\x04\x00", 3);
        WriteMemory((LPVOID)(dwEHSVC + 0x0085B43), (LPVOID)"\xC3", 1);
        WriteMemory((LPVOID)(dwEHSVC + 0x000A238), (LPVOID)"\x74", 1);
    WriteMemory((LPVOID)(dwEHSVC + 0x008523E), (LPVOID)"\xC2\x04\x00", 3);
    WriteMemory((LPVOID)(dwEHSVC + 0x00A5EBA), (LPVOID)"\xD2", 1);
}
```

Xtrap Bypass Author: Slicktor

C++:

```
#include "Bypass.h"

DWORD WINAPI InitializeXTrapBypass() {

    DWORD nBase;
    while(1)
    {

        nBase = (DWORD)GetModuleHandleA("XTrapVa.dll");

        if(nBase){
        Sleep(500);
        BYPASS bypass;
        bypass.Driver64();
        bypass.ProcessDetection();
        break;
        }

    }
    return 0;

}

BOOL WINAPI DllMain ( HMODULE hDll, DWORD dwReason, LPVOID lpReserved )
{
    DisableThreadLibraryCalls(hDll);
    if( dwReason == DLL_PROCESS_ATTACH)

    {


_beginthread((void(*)(void*))InitializeXTrapBypass,sizeof(&InitializeXTrapBypass),0);
    }


    return TRUE;
}

//main.cpp

#include <Windows.h>
#include <tlhelp32.h>
#include <process.h>
```

```
#include <wchar.h>

class BYPASS
{

public:
int BYPASS::ProcessDetection();
int BYPASS::Driver64();
};
int BYPASS::ProcessDetection()
{

    DWORD K32EnumAddr =
(DWORD)GetProcAddress(LoadLibraryA("Kernel32.dll"),"K32EnumProcesses");
    //DWORD EnumAddr = (DWORD)GetProcAddress(LoadLibraryA("Psapi.dll"),"EnumProcesses");
    DWORD old;
    VirtualProtect((LPVOID)K32EnumAddr,sizeof(K32EnumAddr),PAGE_EXECUTE_READWRITE,&old);
    //VirtualProtect((LPVOID)EnumAddr,sizeof(EnumAddr),PAGE_EXECUTE_READWRITE,&old);
    memcpy((LPVOID)K32EnumAddr,(LPVOID)"\xC2\x0C\x00",3);
    //memcpy((LPVOID)EnumAddr,(LPVOID)"\xC2\x0C\x00",3);
    return 0;
}

int BYPASS::Driver64()
{

    wmemcpy((wchar_t*)0x405D0C24,(const wchar_t*)"X6va01",6);
    return 0;
}
```

Another bypass source code:

C++:

```
DWORD XTrapDriver = 0x40A20840;

int ThreadDetection()
{
    DWORD oldprotect = 0;
    DWORD K32EnumAddr = (DWORD)GetProcAddress(LoadLibraryA("Kernel32.dll"),
"K32EnumProcesses");
    VirtualProtect((LPVOID)K32EnumAddr, sizeof(K32EnumAddr), PAGE_EXECUTE_READWRITE,
&oldprotect);
    memcpy((LPVOID)K32EnumAddr, (LPVOID)"\xEB\xFE", 2);
    return 0;
}

void Bypass(void*)
{
    while (1)
    {
      DWORD XTrap = (DWORD)GetModuleHandle("XTrapVa.dll");  // get XTrap base address
      HMODULE hwd = GetModuleHandle(TEXT("XTrapVa.dll"));
        if (hwd)// wait XTrapVa.dll
        {
            Sleep(500);
            sHook = (xHook)DetourFunction((PBYTE)XTrapDriver, (PBYTE)Hook);// Hook
            wmemcpy((wchar_t*)sHook, L"X6va02", 6);
            ThreadDetection(); // Call ThreadDetection
            MessageBoxA(NULL, "XTrap Bypass Successful", "Notice", MB_ICONINFORMATION);
            break;
        }
    }
```

```
}

BOOL __stdcall Hook() // Hook
{
    return TRUE;
}


BOOL APIENTRY DllMain(HMODULE hModule,DWORD  ul_reason_for_call,LPVOID lpReserved)
{
    switch (ul_reason_for_call)
    {
    case DLL_PROCESS_ATTACH:
        CreateThread(0, 0, (LPTHREAD_START_ROUTINE)Bypass, 0, 0, 0);
    case DLL_THREAD_ATTACH:
    case DLL_THREAD_DETACH:
    case DLL_PROCESS_DETACH:
        break;
    }
    return TRUE;
}
```

GamersClub Anti-Cheat Information (Driver + user mode module)

Since I haven't posted anything here for a long time, I decided to post my dump + idb for GCSecure.sys, which is the kernel mode driver for GamersClub (CS:GO). Hope someone finds this useful.

Just some quick notes:

-It only registers a notify routine for CreateProcess using PsSetCreateProcessNotifyRoutine that will "wait" for csgo.exe
-Uses ObRegisterCallbacks
-Checks if Disk.sys IoControl Dispatch is hooked, which can be used for hdd serial spoofing. Screenshot
-You can send an IOCTL (Code: 0x2016E040) to the driver to get your process whitelisted. Check the idb for more information about the structures used.

GCSecure.sys virus scan: Antivirus scan for afb12c195b9e343efc51f007379880edffe16ca84f11665493f8a91cf013017e at 2018-10-20 00:59:32 UTC - VirusTotal

GCSECURE_DUMP.sys virus scan:
Antivirus scan for 5f67f4a18367a4aba468cc565cae9978491946e6afb5d136f4fa8079d07ea0e9 at 2018-10-20 01:01:35 UTC - VirusTotal

Dumped their user mode module yesterday. Still reversing it but found some interesting things:

-They hook LdrLoadDll to block module loading (could be done from their driver tho. Guess they are just dumb)

(They call GetProcAddress for LdrInitializeThunk but don't do any shit)

```
  v1 = LoadLibraryA("ntdll.dll");
  sub_10016740();
  o_LdrLoadDll = (int (__stdcall *)(_DWORD, _DWORD, _DWORD, _DWORD))GetProcAddress(v1, "LdrLoadDll");
  GetProcAddress(v1, "LdrInitializeThunk");
  sub_10001250();
  v2 = GetCurrentThread();
  sub_10001AA0(v2);
  set_hook((int)&o_LdrLoadDll, hk_LdrLoadDll);
  sub_100014B0();
  GrabFuncs();
  sub_1001A160();
}
```

Whitelisted modules:
C++:

```
int Trust_Dlls()
{
  PUNICODE_STRING v0; // ecx
  PUNICODE_STRING v1; // ecx
  PUNICODE_STRING v2; // ecx
  PUNICODE_STRING v3; // ecx
  PUNICODE_STRING v4; // ecx
  PUNICODE_STRING v5; // ecx
```

```
PUNICODE_STRING v6; // ecx
PUNICODE_STRING v7; // ecx
PUNICODE_STRING v8; // ecx
PUNICODE_STRING v9; // ecx
PUNICODE_STRING v10; // ecx
PUNICODE_STRING v11; // ecx
PUNICODE_STRING v12; // ecx
PUNICODE_STRING v13; // ecx
PUNICODE_STRING v14; // ecx
PUNICODE_STRING v15; // ecx
PUNICODE_STRING v16; // ecx
PUNICODE_STRING v17; // ecx
PUNICODE_STRING v18; // ecx
PUNICODE_STRING v19; // ecx
PUNICODE_STRING v20; // ecx
PUNICODE_STRING v21; // ecx
PUNICODE_STRING v22; // ecx
PUNICODE_STRING v23; // ecx
PUNICODE_STRING v24; // ecx
PUNICODE_STRING v25; // ecx
PUNICODE_STRING v26; // ecx
int v27; // ecx
int v29; // [esp+4h] [ebp-29Ch]
__int16 v30; // [esp+8h] [ebp-298h]
int v31; // [esp+18h] [ebp-288h]
int v32; // [esp+1Ch] [ebp-284h]
__int16 v33; // [esp+20h] [ebp-280h]
int v34; // [esp+30h] [ebp-270h]
int v35; // [esp+34h] [ebp-26Ch]
__int16 v36; // [esp+38h] [ebp-268h]
int v37; // [esp+48h] [ebp-258h]
int v38; // [esp+4Ch] [ebp-254h]
__int16 v39; // [esp+50h] [ebp-250h]
int v40; // [esp+60h] [ebp-240h]
int v41; // [esp+64h] [ebp-23Ch]
__int16 v42; // [esp+68h] [ebp-238h]
int v43; // [esp+78h] [ebp-228h]
int v44; // [esp+7Ch] [ebp-224h]
__int16 v45; // [esp+80h] [ebp-220h]
int v46; // [esp+90h] [ebp-210h]
int v47; // [esp+94h] [ebp-20Ch]
__int16 v48; // [esp+98h] [ebp-208h]
int v49; // [esp+A8h] [ebp-1F8h]
int v50; // [esp+ACh] [ebp-1F4h]
__int16 v51; // [esp+B0h] [ebp-1F0h]
int v52; // [esp+C0h] [ebp-1E0h]
int v53; // [esp+C4h] [ebp-1DCh]
__int16 v54; // [esp+C8h] [ebp-1D8h]
int v55; // [esp+D8h] [ebp-1C8h]
int v56; // [esp+DCh] [ebp-1C4h]
__int16 v57; // [esp+E0h] [ebp-1C0h]
int v58; // [esp+F0h] [ebp-1B0h]
int v59; // [esp+F4h] [ebp-1ACh]
__int16 v60; // [esp+F8h] [ebp-1A8h]
int v61; // [esp+108h] [ebp-198h]
int v62; // [esp+10Ch] [ebp-194h]
__int16 v63; // [esp+110h] [ebp-190h]
int v64; // [esp+120h] [ebp-180h]
int v65; // [esp+124h] [ebp-17Ch]
__int16 v66; // [esp+128h] [ebp-178h]
int v67; // [esp+138h] [ebp-168h]
int v68; // [esp+13Ch] [ebp-164h]
__int16 v69; // [esp+140h] [ebp-160h]
```

```
int v70; // [esp+150h] [ebp-150h]
int v71; // [esp+154h] [ebp-14Ch]
__int16 v72; // [esp+158h] [ebp-148h]
int v73; // [esp+168h] [ebp-138h]
int v74; // [esp+16Ch] [ebp-134h]
__int16 v75; // [esp+170h] [ebp-130h]
int v76; // [esp+180h] [ebp-120h]
int v77; // [esp+184h] [ebp-11Ch]
__int16 v78; // [esp+188h] [ebp-118h]
int v79; // [esp+198h] [ebp-108h]
int v80; // [esp+19Ch] [ebp-104h]
__int16 v81; // [esp+1A0h] [ebp-100h]
int v82; // [esp+1B0h] [ebp-F0h]
int v83; // [esp+1B4h] [ebp-ECh]
__int16 v84; // [esp+1B8h] [ebp-E8h]
int v85; // [esp+1C8h] [ebp-D8h]
int v86; // [esp+1CCh] [ebp-D4h]
__int16 v87; // [esp+1D0h] [ebp-D0h]
int v88; // [esp+1E0h] [ebp-C0h]
int v89; // [esp+1E4h] [ebp-BCh]
__int16 v90; // [esp+1E8h] [ebp-B8h]
int v91; // [esp+1F8h] [ebp-A8h]
int v92; // [esp+1FCh] [ebp-A4h]
__int16 v93; // [esp+200h] [ebp-A0h]
int v94; // [esp+210h] [ebp-90h]
int v95; // [esp+214h] [ebp-8Ch]
__int16 v96; // [esp+218h] [ebp-88h]
int v97; // [esp+228h] [ebp-78h]
int v98; // [esp+22Ch] [ebp-74h]
__int16 v99; // [esp+230h] [ebp-70h]
int v100; // [esp+240h] [ebp-60h]
int v101; // [esp+244h] [ebp-5Ch]
__int16 v102; // [esp+248h] [ebp-58h]
int v103; // [esp+258h] [ebp-48h]
int v104; // [esp+25Ch] [ebp-44h]
__int16 v105; // [esp+260h] [ebp-40h]
int v106; // [esp+270h] [ebp-30h]
int v107; // [esp+274h] [ebp-2Ch]
__int16 v108; // [esp+278h] [ebp-28h]
int v109; // [esp+288h] [ebp-18h]
int v110; // [esp+28Ch] [ebp-14h]
int v111; // [esp+290h] [ebp-10h]
int v112; // [esp+29Ch] [ebp-4h]

sub_100161E0();
v32 = 7;
v31 = 0;
v30 = 0;
sub_10017650(v0, L"\\system32\\uxtheme.dll", 21);
sub_100161E0();
v35 = 7;
v34 = 0;
v33 = 0;
sub_10017650(v1, L"\\system32\\user32.dll", 20);
sub_100161E0();
v38 = 7;
v37 = 0;
v36 = 0;
sub_10017650(v2, L"\\system32\\winrnr.dll", 20);
sub_100161E0();
v41 = 7;
v40 = 0;
v39 = 0;
```

```
sub_10017650(v3, L"\\system32\\fwpucInt.dll", 22);
sub_100161E0();
v44 = 7;
v43 = 0;
v42 = 0;
sub_10017650(v4, L"\\system32\\rasadhlp.dll", 22);
sub_100161E0();
v47 = 7;
v46 = 0;
v45 = 0;
sub_10017650(v5, L"\\system32\\windows.ui.dll", 24);
sub_100161E0();
v50 = 7;
v49 = 0;
v48 = 0;
sub_10017650(v6, L"\\system32\\dsound.dll", 20);
sub_100161E0();
v53 = 7;
v52 = 0;
v51 = 0;
sub_10017650(v7, L"\\system32\\rsaenh.dll", 20);
sub_100161E0();
v56 = 7;
v55 = 0;
v54 = 0;
sub_10017650(v8, L"\\system32\\crypt32.dll", 21);
sub_100161E0();
v59 = 7;
v58 = 0;
v57 = 0;
sub_10017650(v9, L"\\system32\\wintrust.dll", 22);
sub_100161E0();
v62 = 7;
v61 = 0;
v60 = 0;
sub_10017650(v10, L"\\system32\\mswsock.dll", 21);
sub_100161E0();
v65 = 7;
v64 = 0;
v63 = 0;
sub_10017650(v11, L"\\system32\\ole32.dll", 19);
sub_100161E0();
v68 = 7;
v67 = 0;
v66 = 0;
sub_10017650(v12, L"\\system32\\gdi32.dll", 19);
sub_100161E0();
v71 = 7;
v70 = 0;
v69 = 0;
sub_10017650(v13, L"\\system32\\wshtcpip.dll", 22);
sub_100161E0();
v74 = 7;
v73 = 0;
v72 = 0;
sub_10017650(v14, L"\\system32\\shell32.dll", 21);
sub_100161E0();
v77 = 7;
v76 = 0;
v75 = 0;
sub_10017650(v15, L"\\system32\\advapi32.dll", 22);
sub_100161E0();
v80 = 7;
```

```
v79 = 0;
v78 = 0;
sub_10017650(v16, L"\\system32\\kernel32.dll", 22);
sub_100161E0();
v83 = 7;
v82 = 0;
v81 = 0;
sub_10017650(v17, L"\\system32\\msctf.dll", 19);
sub_100161E0();
v86 = 7;
v85 = 0;
v84 = 0;
sub_10017650(v18, L"\\system32\\bcryptprimitives.dll", 30);
sub_100161E0();
v89 = 7;
v88 = 0;
v87 = 0;
sub_10017650(v19, L"\\system32\\advapi32.dll", 22);
sub_100161E0();
v92 = 7;
v91 = 0;
v90 = 0;
sub_10017650(v20, L"\\system32\\gpapi.dll", 19);
sub_100161E0();
v95 = 7;
v94 = 0;
v93 = 0;
sub_10017650(v21, L"\\system32\\cryptsp.dll", 21);
sub_100161E0();
v98 = 7;
v97 = 0;
v96 = 0;
sub_10017650(v22, L"\\system32\\hssrv.dll", 19);
sub_100161E0();
v101 = 7;
v100 = 0;
v99 = 0;
sub_10017650(v23, L"\\system32\\igc32.dll", 19);
sub_100161E0();
v104 = 7;
v103 = 0;
v102 = 0;
sub_10017650(v24, L"\\syswow64\\wintrust.dll", 22);
sub_100161E0();
v107 = 7;
v106 = 0;
v105 = 0;
sub_10017650(v25, L"\\syswow64\\crypt32.dll", 21);
sub_100161E0();
v110 = 7;
v109 = 0;
v108 = 0;
sub_10017650(v26, L"\\syswow64\\bcryptprimitives.dll", 30);
v112 = 26;
sub_100185F0(v27);
LOBYTE(v29) = 0;
dword_100B5000 = 0;
dword_100B5004 = 0;
dword_100B5008 = 0;
Alloc_Vector((int)&v30, (int)&v111, v29);
v112 = -1;
`eh vector destructor iterator'(&v30, 0x18u, 0x1Bu, sub_10017340);
return atexit((void (__cdecl *)())sub_10098B20);
```

```
}
```

Just manual map your shit and you're good

-They use D3DXSaveSurfaceToFileInMemory to take screenshots (they analyze the screenshots to see if someone is cheating).

I'm attaching the dll to this post if someone wants to reverse it

How to Bypass Kernel Anticheat & Develop Drivers

All popular games are utilizing kernel anticheat, in this cat and mouse game, the hackers must now enter kernel mode as well. Kernel Anticheat is very effective in preventing usermode cheats. This guide will provide you everything you need to know to start learning how to bypass kernel anticheat. If you have not finished the Guided Hacking Bible, do not waste your time on kernel anticheat, you're not ready.

The information provided in this guide will cover:

- Kernel Mode vs Usermode
- How to learn kernel driver development
- A video tutorial series covering kernel mode cheats
- How to exploit vulnerable drivers
- Common vulnerable drivers & tools
- An overview of the common functionality of kernel anticheats
- Detection of kernel cheats

Anticheats Utilizing Kernel Modules

BattleEye, Xigncode, Easy Anti Cheat, Vanguard

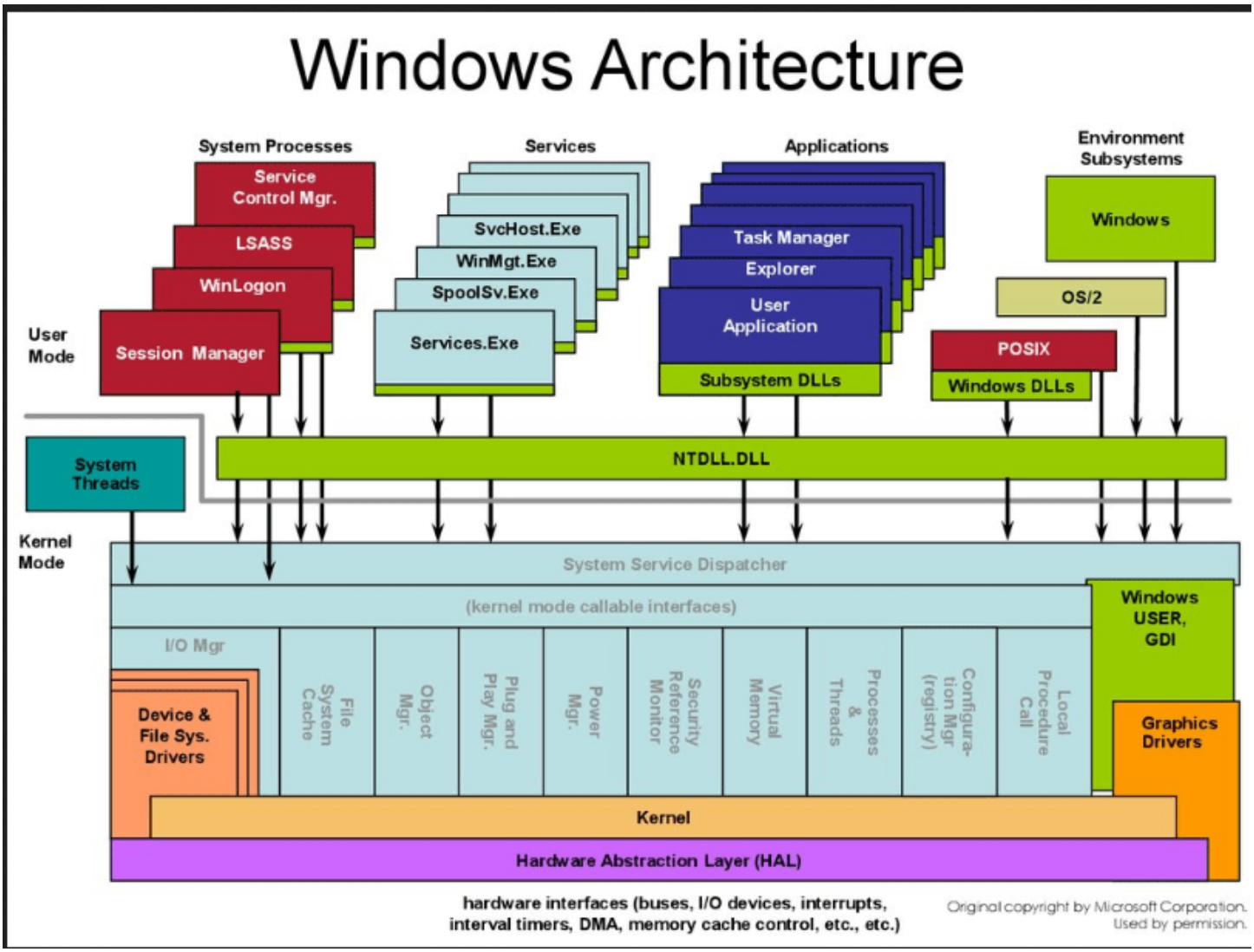
What is a kernel mode driver & Kernel Mode vs User Mode

A processor in a Windows computer has two different modes: *kernel mode* and user *mode*. The processor switches between the two modes depending on what type of code is running. Normal .exe programs run in user mode & core operating system components run in kernel mode. The Usermode & Kernelmode construct is built into the CPU. The low level core functionality of the operating system is done in kernel mode, which is a privileged part of memory that is not accessible from user mode and executes with privileged status on the CPU. Drivers are not just limited to Hardware Drivers, you can make a .sys driver to do anything you want in kernel mode, including bypass anticheat and perform cheat functionality.

A user mode process resides in it's own personal virtual address space that is private and doesn't interact with other processes's memory normally. Each application runs in isolation, if a regular program crashes, the crash is limited to that one application. Other applications and the operating system are not affected by the crash.

All code that runs in kernel mode shares a single virtual address space. This means that a kernel-mode driver is not isolated from other drivers and the operating system itself. If a kernel-mode driver accidentally writes to the wrong virtual address, data that belongs to the operating system or another driver could be compromised. If a kernel-mode driver crashes, the entire operating system crashes.

Many of the privelages things you need to do in game hacking rely on the kernel performing those tasks for you. When you call WriteProcessMemory() for example, that function is exported by NTDLL.DLL and that request to write to the memory of another process is passed on to the kernel through NTDLL. You application isn't actually doing it, the kernel is, your program is just making the request. View the image below to understand how kernel mode and usermode are separated.



User mode processes don't have access to kernel mode processes and memory. That is how the CPU and Operating System are designed.

How does this apply to bypassing Anticheat?

If you are dealing with a strong usermode anticheat, you can write a kernel mode driver to bypass it. Because you are in the kernel and the anticheat is not, you can modify the anticheat to stop it's detection or you can hide your usermode module from it entirely. A user mode anticheat has no idea what you're doing in kernel.

If the anticheat has a kernel driver then you must also be in kernel mode, because nothing you do in usermode is going to be able to bypass or hide from a kernel anticheat. Generally speaking, kernel mode drivers are not necessary to hack 99% of games. In fact, kernel mode drivers are very easy to detect by anticheat if not done correctly.

Coding a kernel driver is much more complicated than user mode applications, for which reason your functionality which provides the "bypass" is done in the kernel but in most cases, the actual cheat logic is done in a usermode module. In this situation, you load your driver, enable your "bypass" functionality and then inject your DLL. Alternatively you can write your entire hack to run in kernel mode, which is more difficult.

But Rake, I don't want to learn, I just want to paste some crap and bypass anticheat!

Ok before we go to far I will give you a simple 6 step process that is the easiest way to paste your way into kernel:

1. Video Tutorial - How to Make a Windows Kernel Mode Driver Tutorial
2. Video Tutorial - Kernel 2 - Usermode Communication - IOCTL Tutorial
3. Video Tutorial - How to Write Memory from Kernel - MmCopyVirtualMemory Tutorial
4. Experiment with this source code Source Code - CSGO Kernel Driver Multihack
5. Use kdmapper which uses a vulnerable Intel driver to manually map your kernel driver (make sure anticheat is not loaded yet)
6. Start the game and use your usermode application to write to the game memory

With those 5 steps, you can start writing to the memory of games with anticheat. But EAC and other strong kernel anticheats can detect this easily, so keep reading to learn more.

Kernel Driver Development

To get started with driver development start with these resources:

- Video Tutorial - How to Make a Windows Kernel Mode Driver Tutorial
- User mode and kernel mode - Windows drivers
- Kernel-Mode Driver Architecture Design Guide - Windows drivers
- Getting started with Windows drivers
- Download the Windows Driver Kit (WDK)
- Windows Driver Development - Windows Hardware Dev Center
- Write a universal Hello World driver (KMDF)

Driver Signing & Test Signing

Windows security would certainly be lacking if you could just load any kernel driver you wanted. This is why Windows requires your kernel mode driver to be signed with a security certificate in order for the OS to load it, but don't worry you don't need to pay 200$ for a certificate. You need to enable Test Signing if you want to load a driver you're actively developing.

In the past you could disable Driver Signing by running these commands as admin and rebooting:

C++:

```
bcdedit.exe -set loadoptions DDISABLE_INTEGRITY_CHECKS
bcdedit.exe -set TESTSIGNING ON
```

On Windows 8 and 10 you may need to do this by accessing the Advanced Boot Options menu by pressing F8 during boot. Windows 10 has disabled the F8 hotkey, to re-enable it:

C++:

```
bcdedit /set {default} bootmenupolicy legacy
```

Then reboot, and press F8 before Windows loads and you will see a menu in which you can Disable Driver Signing. Alternatively on Windows 10 you can hold SHIFT when you click Restart, and this menu will appear. But it only works for that one reboot, you need to do it every time because Windows 10 resets it back to default value.

Kernel Anticheats Prevent games from loading when Test Signing is enabled

The kernel anticheat developers got wise to this, and now they prevent you from playing the game if Test Signing is enabled. So you're forced to enable Driver Signing.

Then how do you load your driver? Keep reading my young padawan.

Exploiting Kernel Drivers

Kernel drivers are very common not just for hardware drivers, many different types of software utilize them. Driver security is very poor and there are many vulnerable drivers. The drivers expose functions to their usermode applications, to make development easy and cheap, they often expose too much or provide functionality that is too dangerous.

Any driver that takes data from usermode and does something with it in kernel is potentially vulnerable. Many have buffer overflows which can be leveraged, or even worse an arbitrary kernel write vulnerability. These vulnerabilities can be exploited from usermode to execute your code, ideally providing a simple method to load your own driver.

But you can't just load your driver, you need to manually map it because it is not digitally signed. These vulnerable kernel drivers must have valid security certificates. By utilizing a valid & certified driver, you can manually map your unsigned driver without issue. Microsoft or the Certificate Authorities can decide to reject these certificates at any time, making them no longer work, but that is extremely rare.

For learning purposes learn to use KDMapper first, and then learn how to use KDU

KDMapper

KDMapper is used by hundreds of pay cheat providers and for good reason, it's super paste friendly.

- Utilizes an embedded vulnerable Intel driver
- Manually Maps your driver
- Provides a simple command line interface
- You just pass it 1 argument and you're driver is loaded

KDMapper comes embedded with the vulnerable iqvw64e.sys Intel Ethernet diagnostics driver driver. The driver is embedded as a byte array in intel_driver_resource.hpp

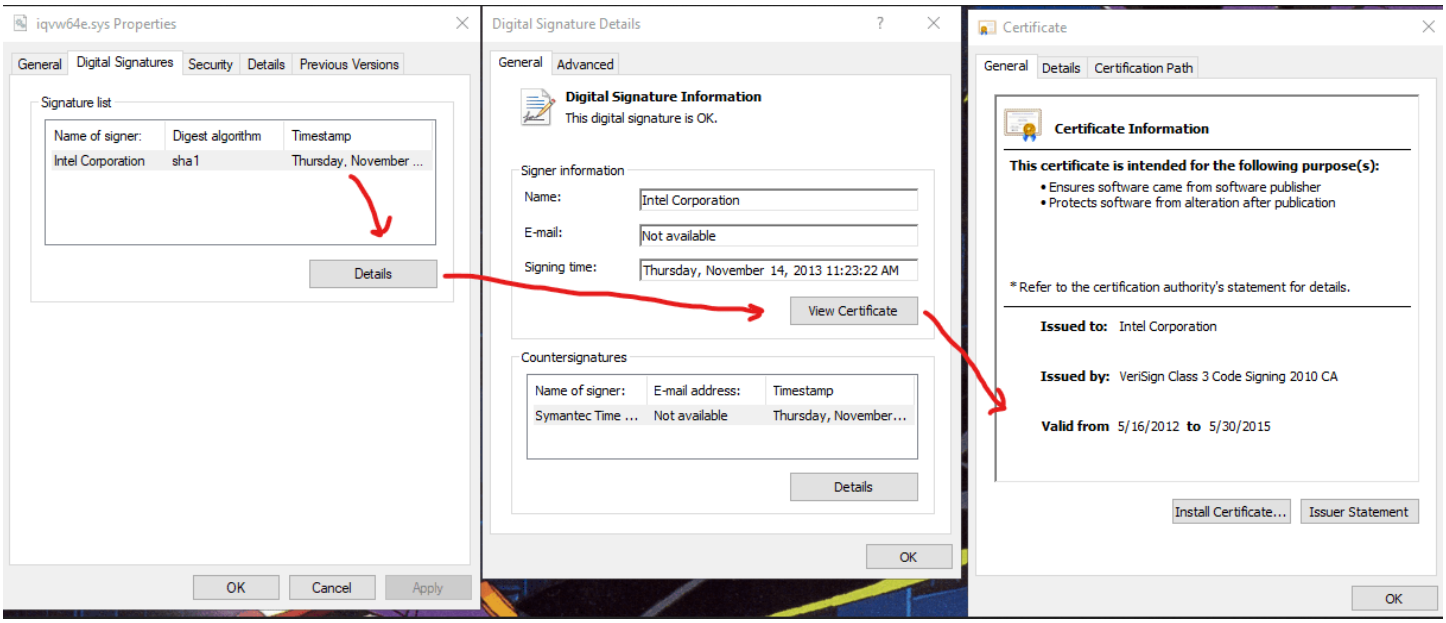
The driver was signed in 2013. The vulnerability was officially published in 2015 as CVE 2015 2291 with a severity score of 7.8. Amazingly it's certificate has not been revoked yet.

iqvw64e.sys
Code:

```
sha256            :     B2B2A748EA3754C90C83E1930336CF76C5DF9CBB1E3EEC175164BB01A54A4701
date              :     empty
language          :     English-United States
code-page         :     Unicode UTF-16    :      little endian
CompanyName       :       Intel Corporation
FileDescription   :       Intel(R) Network Adapter Diagnostic Driver
FileVersion       :       1.03.0.7 built by WinDDK
InternalName      :     iQVW64.SYS
LegalCopyright    :       Copyright (C) 2002-2013 Intel Corporation All Rights Reserved.
OriginalFilename:       iQVW64.SYS
ProductName       :       Intel(R) iQVW64.SYS
```

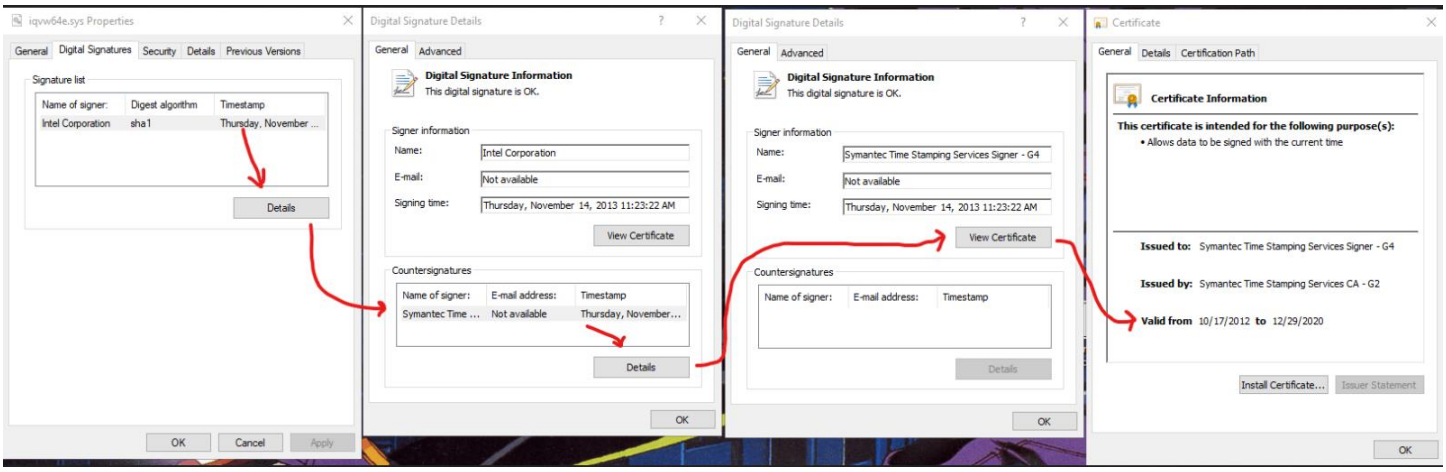
```
ProductVersion    :    1.03.0.7
```

iqvw64e.sys Main Intel Signature



But wait it's not valid after 2015! Wrong! Windows still loads it.

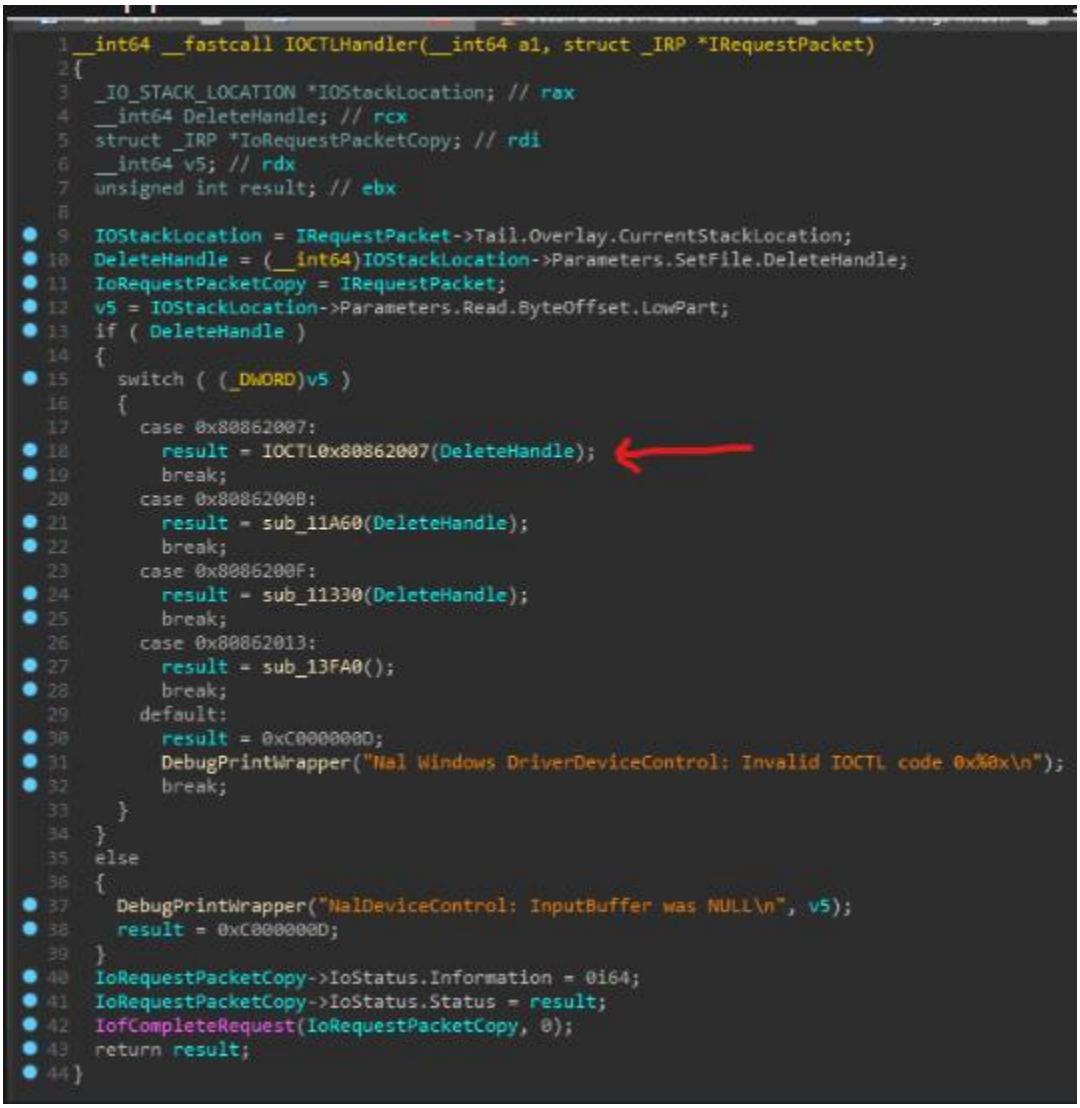
Counter Signer Symantec Time Signature



What happens in December 2020? Nothing! Microsoft will continue to load it as long as it is not revoked!

The vulnerability exists due to insufficient input buffer validation when the driver processes IOCTL codes 0x80862013, 0x8086200B, 0x8086200F, 0x80862007 using METHOD_NEITHER and due to insecure permissions allowing everyone read and write access to privileged use only functionality.

KdMapper utilizes IOCTL code 0x80862007 for arbitrary kernel execute



```
__int64 __fastcall IOCTLHandler(__int64 a1, struct _IRP *IRequestPacket)
{
  _IO_STACK_LOCATION *IOStackLocation; // rax
  __int64 DeleteHandle; // rcx
  struct _IRP *IoRequestPacketCopy; // rdi
  __int64 v5; // rdx
  unsigned int result; // ebx

  IOStackLocation = IRequestPacket->Tail.Overlay.CurrentStackLocation;
  DeleteHandle = (__int64)IOStackLocation->Parameters.SetFile.DeleteHandle;
  IoRequestPacketCopy = IRequestPacket;
  v5 = IOStackLocation->Parameters.Read.ByteOffset.LowPart;
  if ( DeleteHandle )
  {
    switch ( (_DWORD)v5 )
    {
      case 0x80862007:
        result = IOCTL0x80862007(DeleteHandle);
        break;
      case 0x8086200B:
        result = sub_11A60(DeleteHandle);
        break;
      case 0x8086200F:
        result = sub_11330(DeleteHandle);
        break;
      case 0x80862013:
        result = sub_13FA0();
        break;
      default:
        result = 0xC000000D;
        DebugPrintWrapper("Nal Windows DriverDeviceControl: Invalid IOCTL code 0x%0x\n");
        break;
    }
  }
  else
  {
    DebugPrintWrapper("NalDeviceControl: InputBuffer was NULL\n", v5);
    result = 0xC000000D;
  }
  IoRequestPacketCopy->IoStatus.Information = 0i64;
  IoRequestPacketCopy->IoStatus.Status = result;
  IofCompleteRequest(IoRequestPacketCopy, 0);
  return result;
}
```

KDMapper is very easy to detect by anticheat - The driver is well documented, everyone knows what it is. But it's a good start to get you exposed to kernel hacking. Read more @ Download - KDMapper - Manually Map Kernel Drivers CVE-2015-229

List of vulnerable drivers

There are probably thousands of vulnerable drivers, here are some we know about. Learn more about this list @ Discuss - New vulnerable kernel drivers

- iqvw64e.sys
- gpcidrv64.sys
- AsUpIO64.sys
- AsrDrv10.sys
- AsrDrv101.sys
- AsrDrv102.sys
- AsrDrv103.sys
- BSMEMx64.sys
- BSMIXP64.sys
- BSMIx64.sys
- BS_Flash64.sys

- BS_HWMIO64_W10.sys
- BS_HWMIo64.sys
- BS_I2c64.sys
- GLCKIO2.sys
- GVCIDrv64.sys
- HwOs2Ec10x64.sys
- HwOs2Ec7x64.sys
- MsIo64.sys
- NBIOLib_X64.sys
- NCHGBIOS2x64.SYS
- NTIOLib_X64.sys
- PhlashNT.sys
- Phymemx64.sys
- UCOREW64.SYS
- WinFlash64.sys
- WinRing0x64.sys
- amifldrv64.sys
- atillk64.sys
- dbk64.sys
- mtcBSv64.sys
- nvflash.sys
- nvflsh64.sys
- phymem64.sys
- rtkio64.sys
- rtkiow10x64.sys
- rtkiow8x64.sys
- segwindrvx64.sys
- superbmc.sys
- semav6msr.sys
- piddrv64.sys
- RTCore64
- Gdrv
- ATSZIO64
- MICSYS
- GLCKIO2
- EneIo
- WinRing0x64
- EneTechIo

Vulnerable Driver Resources

- Discuss - New vulnerable kernel drivers
- Weaponizing vulnerable driver for privilege escalation— Gigabyte Edition!
- EvanMcBroom/PoCs
- Escaping SMEP Hell: Exploiting Capcom Driver In a Safe Manner
- can1357/safe_capcom
- notscimmy/libcapcom
- Bypassing Anti-Cheats - Part 1 - Exploiting Razer Synapse Driver - Niemand - Cyber Security
- Mother of All Drivers - New Vulnerabilities Found in Windows Drivers - Eclypsium

Everything from hfiref0x is amazing

- hfiref0x - Overview specifically -> hfiref0x/KDU
- The Vault
- @hFireF0X

WOW LOOK AT ME, I BYPASSED KERNEL ANTICHEAT!

You literally did nothing except paste. Stop saying "I have a bypass", you have the same bypass that another 100,000 people are using and all you did was download kdmapper. You're not special, so just shut up please, we're not impressed. Saying "I have a bypass" when you're using kdmapper is like saying "I have Cheat Engine".

General Functionality of Kernel Anticheats

- All the normal usermode detections
- Blocking / stripping of process handles
- Detection of test signing
- Detection of usermode hooks
- Detection of injected modules
- Detection of manually mapped modules
- Detection of kernel drivers
- Detecting of traces of manually mapped drivers
- Detection of virtual machines and emulation

Manually Mapped Driver Detection

You must bypass these things, clear PiDDBCacheTable & MmUnloadedDrivers, and stop the enumeration of your own system pools & threads.

- PiDDBCacheTable & MmUnloadedDrivers
- system pool detection
- system thread detection

Source Code - How to Clear PiDDBCache Table / PiDDBLock

PatchGuard

PatchGuard detects patches in the kernel, you can't just patch the anticheat's kernel driver

What Next?

So you can manually map your driver, and you can read and write memory, what do you do next?

Well you didn't really bypass the anticheat. All you did was load a cheat they didn't detect yet, and now it's very likely they have seen your modules. If the same modules are detected on multiple machines, you may find yourself in the next ban wave. Just making a driver and mapping it doesn't bypass anything. Kernel anticheats are incredibly invasive and they can detect everything that's happening on your system. If you're doing something that looks malicious, they can easily detect it and ban you.

Kernel Anticheat typically are used in combination with a usermode module, which is manually mapped into the game and obfuscated. Your next step is to dump both the kernel module and the usermode module and reverse engineer them. Then you will have a very good idea of how they operate, and what else you need to do completely bypass the anticheat.

Remember, you can't patch the kernel anticheat, so you need to go around it.

Next you want to patch all the usermode detections so you can attach a debugger, especially Cheat Engine & Reclass so you can start reversing the game.

From kernel you can patch or hook all the detection mechanisms in the anticheat's usermode module, and you can use your own kernel module to protect & hide your own usermode module. Essentially you want to block the anticheat from accessing any of your modules address range. Once you've taken care of all of that, you can inject your usermode module without any trouble.

Detection of Kernel Cheats

It's super easy for them to detect vulnerable drivers, the anticheat devs have the same list of vulnerable drivers that we have and they are actively scanning for the most popular ones. If they find your module they will upload it to their server, analyze it and build detection for it.

EAC for example has some very good detection methods, regardless of which anticheat you're trying to bypass you should read our EAC thread to learn more.

A manually mapped driver cannot be detected using the normal methods, but mapping your driver does leave traces behind. Make sure you clear PiDDBCacheTable and anything else your driver leaves behind.

Guided Hacking Kernel Videos

1. Video Tutorial - How to Make a Windows Kernel Mode Driver Tutorial
2. Video Tutorial - Kernel 2 - Usermode Communication - IOCTL Tutorial
3. Video Tutorial - How to Write Memory from Kernel - MmCopyVirtualMemory Tutorial

GH Resources

- Download - KDMapper - Manually Map Kernel Drivers CVE-2015-229
- Guide - How to Bypass EAC - Easy Anti Cheat
- Tutorial - MTA: SA's kernel mode anticheat is a joke (information)
- Guide - Anticheat Battleye Bypass Overview
- Guide - How to bypass XignCode Anticheat Guide - XignCode3
- Source Code - CSGO Kernel Driver Multihack
- Tutorial - MTA: SA's kernel mode anticheat is a joke (information)
- Guide - How anti-cheats detect system emulation
- Download - GamersClub Anti-Cheat Information (Driver + user mode module)

External Resources

- All secret.club articles
- xerox
- hfiref0x/TDL
- hfiref0x/KDU
- hacksysteam/HackSysExtremeVulnerableDriver
- Zer0Mem0ry/ntoskrnl
- FuzzySecurity/Capcom-Rootkit
- tandasat/ExploitCapcom
- SamLarenN/CapcomDKOM
- BlueSkeye/CapcomDriver
- zerosum0x0/ShellcodeDriver

How to Bypass Kernel Anticheat

Test signing via bcdedit still works just dandy

on the same note, while its trivial to detect test signing being enabled in many ways, since youre the kernel, you can attempt to hook their own detections/spoof them and then things work just fine (thats how i used to load my driver vs EAC some years ago)

PatchGuard
==========

In the times of old, everyone and their dead dog would patch the windows kernel, place hooks on whatever APIs they wanted, and this caused lots of system instability when users would download something that decided to put its dick everywhere.

In comes patchguard, microsoft's way of saying "stop f*cking with our OS". So certain modifications will cause (eventually) a BSOD. This includes, but is not limted to: modification of some MSRs (Model specific registers), hooks on certain functions (such as NTAPIs), modification of PatchGuard itself, modification of critical linked lists (such as the EPROCESS list, so you cant hide entire processes from UM enumeration)

Of course, there are ways to disable it, but in every new edition of windows it gets more and more aids. Simple google searches can get you started if thats what youre into.

Development

==========

Im always a big advocate for "try shit and brick stuff", use a VM when coding your drivers so oyu dont brick your acutal PC and can just restore from a snapshot or whatever. Also enables actual debugging of your driver rather than crawling crash dumps.

my main disclaimer for anyone wishing to write a driver. If you ask an issue that i can find an answer to in a single google search then i will ignore you until you show the ability to properly attempt steps of debugging and research.

Example of a good way to ask a question, "Hey, im trying to stop ObRegisterCallbacks in an anticheat and ive noticed that you can try to collide with their altitude. How would one find a specific driver's altitude?"

or

"Hey, i want to stop a driver from loading, ive read that you can do this via LoadImageNotifyRoutines and i've got mine setup. But i dont understand where to go from there."

not

"Hey can you show me how to make a manual mapper in kernel"
"Hi how do i read memory from kernel"

Anticheat XTrap Bypass Source Codes

Introduction - What Is XTrap

X-Trap is an anti-cheating program created and maintained by WiseLogic, used in almost all CrossFire versions to prevent players from using hacking tools.
Like anti-virus programs, X-Trap is launched along with CrossFire and continue manipulating memories while CF is running to detect suspicious processes that try to interact with crossfire.exe. If something goes wrong, X-Trap will close CrossFire and give out an error message, telling players the possible reasons and general suggestions to fix it. Accessing process viewers, like Task Manager, is also counted as suspicious activities (This is X-Trap's self-defense module to avoid being "killed").

Though not as effective as many people think, X-Trap is actually better at detecting and blocking hack tools, some that GameGuard can't detect which was the previous anti-cheat program for CrossFire. Naturally, hack tools are updated faster than X-Trap because hackers are everywhere, and they have more resources and have more time, while WiseLogic must work on updating their X-Trap for every publisher, so X-Trap often falls behind when coming to updates. However, X-Trap is still a necessary tool to help protecting CF against popular and public hacks, which many people may use for free.

- To date, CF China, CF North America, CF Brazil and CF Japan does not use X-Trap. CF Japan utilizes Game Guard, CrossFire North America, CF Brazil and CF Español use XIGNCODE3 while CF China has its own anti-cheating program called **Tencent Protect**, which works similar to GameGuard, but acts much more effective, due to in-game file checking. This process requires powerful computers however, so players with decent PCs may have to wait a bit long before the game is loaded.
- In January 2017, CF Brazil changed their anti-cheat from **X-Trap** to **XignCode**.
- In March 2017, CF Español changed their anti-cheat from **X-Trap** to **XignCode**.
- X-Trap can only be run on a computer's administrator account, so it is not possible to play CrossFire in Guest accounts or Standard users' accounts.
- Recent patches in CF Vietnam has X-Trap blacklisted almost all of the auto-clicker programs. This is done to counter event farming, as lots of people have been using auto-clickers to hang in room during events that requires playing a certain amount of times to receive prizes.
- In Feb 2020, CF Philippines changed their anti-cheat from **X-Trap** to **XignCode**.

Other games that use XTrap: Cabal Online, Granado Espada, Tower Alliance Online, Priston Tale
XTrap has many false positives, it will block anything that tries to touch the game process including Process Explorer and other tools.

XTrap Source Code

The XTrap Source code was leaked in 2012 and can be found in the attachments below

XTrap Bypass

Before you try to bypass XTrap you need to learn about anticheat:
Guide - How to Bypass Anticheat - Start Here Beginner's Guide

S4League Old Outdated Bypass

available in the attachments

Various Bypasses

Bypass 1

C++:

```
DWORD XTrapDriver = 0x40A20840;

int ThreadDetection()
{
    DWORD oldprotect = 0;
    DWORD K32EnumAddr = (DWORD)GetProcAddress(LoadLibraryA("Kernel32.dll"),
"K32EnumProcesses");
    VirtualProtect((LPVOID)K32EnumAddr, sizeof(K32EnumAddr), PAGE_EXECUTE_READWRITE,
&oldprotect);
    memcpy((LPVOID)K32EnumAddr, (LPVOID)"\xEB\xFE", 2);
    return 0;
}

void Bypass(void*)
{
    while (1)
    {
      DWORD XTrap = (DWORD)GetModuleHandle("XTrapVa.dll");  // get XTrap base address
      HMODULE hwd = GetModuleHandle(TEXT("XTrapVa.dll"));
        if (hwd)// wait XTrapVa.dll
        {
            Sleep(500);
            sHook = (xHook)DetourFunction((PBYTE)XTrapDriver, (PBYTE)Hook);// Hook
            wmemcpy((wchar_t*)sHook, L"X6va02", 6);
            ThreadDetection(); // Call ThreadDetection
            MessageBoxA(NULL, "XTrap Bypass Successful", "Notice", MB_ICONINFORMATION);
            break;
        }
    }
}

BOOL __stdcall Hook() // Hook
{
    return TRUE;
}


BOOL APIENTRY DllMain(HMODULE hModule,DWORD  ul_reason_for_call,LPVOID lpReserved)
{
    switch (ul_reason_for_call)
    {
    case DLL_PROCESS_ATTACH:
```

```
        CreateThread(0, 0, (LPTHREAD_START_ROUTINE)Bypass, 0, 0, 0);
    case DLL_THREAD_ATTACH:
    case DLL_THREAD_DETACH:
    case DLL_PROCESS_DETACH:
        break;
    }
    return TRUE;
}
```

Bypass 2

Code:

```
Module Hook

#Region "Access"
    'Setting some privileges.
    Const PROCESS_ALL_ACCESS = &H1F0FF

    Public Enum ThreadAccess As Integer
        TERMINATE = (&H1)
        SUSPEND_RESUME = (&H2)
        GET_CONTEXT = (&H8)
        SET_CONTEXT = (&H10)
        SET_INFORMATION = (&H20)
        QUERY_INFORMATION = (&H40)
        SET_THREAD_TOKEN = (&H80)
        IMPERSONATE = (&H100)
        DIRECT_IMPERSONATION = (&H200)
    End Enum
#End Region

#Region "Functions"
    Public Declare Function OpenProcess Lib "kernel32" (ByVal dwDesiredAccess As Integer,
ByVal bInheritHandle As Integer, ByVal dwProcessId As Integer) As Integer

    'Functions that will allow us to write/read process memory.
    Public Declare Function WriteProcessMemory1 Lib "kernel32" Alias "WriteProcessMemory"
(ByVal hProcess As Integer, ByVal lpBaseAddress As Integer, ByRef lpBuffer As Integer,
ByVal nSize As Integer, ByRef lpNumberOfBytesWritten As Integer) As Integer
    Public Declare Function ReadProcessMemory1 Lib "kernel32" Alias "ReadProcessMemory"
(ByVal hProcess As Integer, ByVal lpBaseAddress As Integer, ByRef lpBuffer As Integer,
ByVal nSize As Integer, ByRef lpNumberOfBytesWritten As Integer) As Integer

    'Functions to suspend/resume the process.
    Public Declare Function OpenThread Lib "kernel32.dll" (ByVal dwDesiredAccess As
ThreadAccess, ByVal bInheritHandle As Boolean, ByVal dwThreadId As UInteger) As IntPtr
    Public Declare Function SuspendThread Lib "kernel32.dll" (ByVal hThread As IntPtr) As
UInteger
    Public Declare Function ResumeThread Lib "kernel32.dll" (ByVal hThread As IntPtr) As
UInteger
    Public Declare Function CloseHandle Lib "kernel32.dll" (ByVal hHandle As IntPtr) As
Boolean
#End Region

#Region "Suspend/Resume"
    'Some functions that allow us to suspend/resume the process.
    Public Function SuspendProcess(ByVal nProcess As System.Diagnostics.Process)
        For Each t As ProcessThread In nProcess.Threads
            Dim th As IntPtr

            th = OpenThread(ThreadAccess.SUSPEND_RESUME, False, t.Id)
```

```
            If th <> IntPtr.Zero Then
                SuspendThread(th)
                CloseHandle(th)
            End If
        Next
    End Function

    Public Function ResumeProcess(ByVal nProcess As System.Diagnostics.Process)
        For Each t As ProcessThread In nProcess.Threads
            Dim th As IntPtr

            th = OpenThread(ThreadAccess.SUSPEND_RESUME, False, t.Id)

            If th <> IntPtr.Zero Then
                ResumeThread(th)
                CloseHandle(th)
            End If
        Next
    End Function
#End Region

#Region "Memory"
    Public Function GetMemoryAddress(ByVal nProcess As String, ByVal nBaseAddress As
Integer, ByVal nOffsets As Integer(), ByVal nLevel As Integer, Optional ByVal nSize As
Integer = 4) As Integer
        Dim nAddress As Integer = nBaseAddress

        For i As Integer = 1 To nLevel
            nAddress = ReadInteger(nProcess, nAddress, nSize) + nOffsets(i - 1)
        Next

        Return nAddress
    End Function

    Public Function ReadInteger(ByVal nProcess As String, ByVal nAddress As Integer,
Optional ByVal nSize As Integer = 4) As Integer
        If nProcess.EndsWith(".exe") Then
            nProcess = nProcess.Replace(".exe", Nothing)
        End If

        Dim ProcessHandle As Process() = Process.GetProcessesByName(nProcess)

        If Not ProcessHandle.Count = 1 Then
            Exit Function
        End If

        Dim hProcess As IntPtr = OpenProcess(PROCESS_ALL_ACCESS, 0, ProcessHandle(0).Id)

        If hProcess = IntPtr.Zero Then
            Exit Function
        End If

        Dim hAddress As Integer
        Dim vBuffer As Integer
        hAddress = nAddress

        ReadProcessMemory1(hProcess, hAddress, vBuffer, nSize, 0)

        Return vBuffer
    End Function
```

```
    Public Function DefineBytes(ByVal nProcess As String, ByVal nAddress As Integer,
ByVal nValue As String)
        If nProcess.EndsWith(".exe") Then
            nProcess = nProcess.Replace(".exe", Nothing)
        End If

        If nValue.Contains(" ") Then
            nValue = nValue.Replace(" ", Nothing)
        End If

        Dim ProcessHandle As Process() = Process.GetProcessesByName(nProcess)

        If ProcessHandle.Length = 0 Then
            Exit Function
        End If

        Dim hProcess As IntPtr = OpenProcess(PROCESS_ALL_ACCESS, 0, ProcessHandle(0).Id)

        If hProcess = IntPtr.Zero Then
            Exit Function
        End If

        Dim C As Integer
        Dim B As Integer
        Dim D As Integer
        Dim V As Byte

        B = 0
        D = 1
        For C = 1 To Math****und((Len(nValue) / 2))
            V = Val("&H" & Mid$(nValue, D, 2))
            Call WriteProcessMemory1(hProcess, nAddress + B, V, 1, 0&)
            B = B + 1
            D = D + 2
        Next C
    End Function
#End Region

#Region "Message(s)"
    'Some defines.
    Dim Credits As String = ("This bypass was created by Papulatus, happy hacking! ^^")
REM: You could just leech this bypass, but I would appreciate it if you credit me :).
    Dim Bit32 As String = ("This bypass doesn't support 32-Bit!") REM: Disappoint some
32-Bit users.
    Dim SearchFailed As String = ("Couldn't find the MicroVolts directory, please put
this application in the 'Bin' folder of MicroVolts!") REM: Message to display if we
couldn't find the MicroVolts directory.
#End Region

#Region "Required addresses"
    'The addresses we'll need to bypass XTrap.
    Dim GetProcAddress As Integer
    Dim ReadProcessMemory As Integer
    Dim XTrapDriver As Integer
#End Region

#Region "Timer(s)"
    Dim MainTMR As New System.Timers.Timer REM: Timer to do some important stuff.
#End Region

#Region "Main" REM: Our main.
    Sub Main()
        'Timer settings:
```

```
        MainTMR.AutoReset = True
        MainTMR.Interval = 1
        AddHandler MainTMR.Elapsed, AddressOf MainTMR_Tick

        If Environment.Is64BitOperatin****tem = False Then REM: Detect 32-Bit users.
            Console.WriteLine(Bit32)
        Else
            If My.Computer.FileSystem.CurrentDirectory.Contains("\MicroVolts\Bin") Then
REM: Check if the application is in the 'Bin' folder of MicroVolts.
                Console.WriteLine(Credits)
                My.Computer.FileSystem.CurrentDirectory =
My.Computer.FileSystem.CurrentDirectory.Replace("\Bin", Nothing) REM: Set current
directory.
                Process.Start("Bin\MicroVolts.exe")
                MainTMR.Start()
            Else
                If My.Computer.FileSystem.DirectoryExists("C:\Program Files\MicroVolts\")
Then
                    Console.WriteLine(Credits)
                    My.Computer.FileSystem.CurrentDirectory = ("C:\Program
Files\MicroVolts\") REM: Set current directory.
                    Process.Start("Bin\MicroVolts.exe")
                    MainTMR.Start()
                ElseIf My.Computer.FileSystem.DirectoryExists("C:\Program Files
(x86)\MicroVolts\") Then
                    Console.WriteLine(Credits)
                    My.Computer.FileSystem.CurrentDirectory = ("C:\Program Files
(x86)\MicroVolts\") REM: Set current directory.
                    Process.Start("Bin\MicroVolts.exe")
                    MainTMR.Start()
                ElseIf My.Computer.FileSystem.DirectoryExists("C:\Archivos de
Programa\MicroVolts\") Then
                    Console.WriteLine(Credits)
                    My.Computer.FileSystem.CurrentDirectory = ("C:\Archivos de
Programa\MicroVolts\") REM: Set current directory.
                    Process.Start("Bin\MicroVolts.exe")
                    MainTMR.Start()
                ElseIf My.Computer.FileSystem.DirectoryExists("C:\Archivos de Programa
(x86)\MicroVolts\") Then
                    Console.WriteLine(Credits)
                    My.Computer.FileSystem.CurrentDirectory = ("C:\Archivos de Programa
(x86)\MicroVolts\") REM: Set current directory.
                    Process.Start("Bin\MicroVolts.exe")
                    MainTMR.Start()
                ElseIf My.Computer.FileSystem.DirectoryExists("C:\MicroVolts\") Then
                    Console.WriteLine(Credits)
                    My.Computer.FileSystem.CurrentDirectory = ("C:\MicroVolts\") REM: Set
current directory.
                    Process.Start("Bin\MicroVolts.exe")
                    MainTMR.Start()
                ElseIf My.Computer.FileSystem.DirectoryExists("D:\MicroVolts\") Then
                    Console.WriteLine(Credits)
                    My.Computer.FileSystem.CurrentDirectory = ("D:\MicroVolts\") REM: Set
current directory.
                    Process.Start("Bin\MicroVolts.exe")
                    MainTMR.Start()
                ElseIf My.Computer.FileSystem.DirectoryExists("D:\Program
Files\MicroVolts\") Then
                    Console.WriteLine(Credits)
                    My.Computer.FileSystem.CurrentDirectory = ("D:\Program
Files\MicroVolts\") REM: Set current directory.
                    Process.Start("Bin\MicroVolts.exe")
                    MainTMR.Start()
```

```
                ElseIf My.Computer.FileSystem.DirectoryExists("D:\Program Files
(x86)\MicroVolts\") Then
                    Console.WriteLine(Credits)
                    My.Computer.FileSystem.CurrentDirectory = ("D:\Program Files
(x86)\MicroVolts\") REM: Set current directory.
                    Process.Start("Bin\MicroVolts.exe")
                    MainTMR.Start()
                ElseIf My.Computer.FileSystem.DirectoryExists("D:\Archivos de
Programa\MicroVolts\") Then
                    Console.WriteLine(Credits)
                    My.Computer.FileSystem.CurrentDirectory = ("D:\Archivos de
Programa\MicroVolts\") REM: Set current directory.
                    Process.Start("Bin\MicroVolts.exe")
                    MainTMR.Start()
                ElseIf My.Computer.FileSystem.DirectoryExists("D:\Archivos de Programa
(x86)\MicroVolts\") Then
                    Console.WriteLine(Credits)
                    My.Computer.FileSystem.CurrentDirectory = ("D:\Archivos de Programa
(x86)\MicroVolts\") REM: Set current directory.
                    Process.Start("Bin\MicroVolts.exe")
                    MainTMR.Start()
                Else
                    Console.WriteLine(SearchFailed)
                End If
            End If
        End If

        Do Until Console.Title = (Nothing) REM: A simple infinite loop to keep the
console stay open.
            Console.ReadKey()
        Loop
    End Sub

    Private Sub MainTMR_Tick(ByVal sender As Object, ByVal e As
System.Timers.ElapsedEventArgs)
        Dim MV() As Process = Process.GetProcessesByName("MicroVolts")
        Dim XT() As Process = Process.GetProcessesByName("XTrap.xt")

        GetProcAddress = GetMemoryAddress("MicroVolts", &HF5F0F0, {&H0}, 0, 4) REM: Grab
MicroVolts' GetProcAddress function.
        ReadProcessMemory = ReadInteger("MicroVolts", GetProcAddress, 4) REM: Use
MicroVolts' GetProcAddress function.
        XTrapDriver = GetMemoryAddress("MicroVolts", &H406BECD4, {&H0}, 0, 4) REM: Grab
the XTrap driver.

        'You'll need this if you want to create BYPASSED multiclients.
        Dim MVIndex As Integer = MV.Count - 1
        Dim XTIndex As Integer = XT.Count - 1

        If XT.Count = MV.Count Then REM: Check if XTrap is running.
            'Begin the motherf*cking hook.
            SuspendProcess(MV(MVIndex))
            DefineBytes("MicroVolts", XTrapDriver, "6F 6C 6F 6C 6F 6C 6F") REM: F*cking
up the XTrap driver.
            DefineBytes("MicroVolts", ReadProcessMemory, "EB FE") REM: Send
ReadProcessMemory to an infinite loop.
            ResumeProcess(MV(MVIndex)) REM: Enjoy the bypass ;).

            End REM: Close our handle.
        End If
    End Sub
#End Region
```

```
End Module
```

Bypass 3

C++:

```
/*
Anti TerminateProcess/ExitProcess Check

Description :
     XTrap check the first byte of TerminateProcess/ExitProcess
     if the byte is E9/C2 then XTrap returns true!

What we do :
     xor eax,eax // (so that eax = 0)
     retn
*/
copymemory((void*)(xtrap+0x2C940),(void*)"\x33\xC0\xC3",3);


/*
Anti TerminateProcess/ExitProcess

Description :
     XTrap Closes the process with TermanateProcess first
     if that fails it then trys ExitProcess
     so we just return so that nothing will close our process

What we do :
     return
*/
copymemory((void*)(xtrap+0x31800),(void*)"\xC3\x90\x90\x90\x90",5);


/*
Anti XTrap Message's

Description :
     XTrap likes to be rude and when we playing we get annoying message's
     like Please close program XXXXX so we just return the message kindly ;)

What we do :
     return 8
*/
copymemory((void*)(xtrap+0x388D0),(void*)"\xC2\x08\x00",3);
```

Driver Anti Xtrap by Firefox

C++:

```
/* Replace "dll.h" with the name of your header */
#define _WIN32_WINNT 0x0500
#include <windows.h>
#include <stdio.h>
#include <conio.h>
#include <tlhelp32.h>
#include <shlwapi.h>
#include <iostream>
#include <winioctl.h>

typedef LONG    NTSTATUS;
typedef NTSTATUS (WINAPI *pNtQIT)(HANDLE, LONG, PVOID, ULONG, PULONG);
```

```
#define STATUS_SUCCESS    ((NTSTATUS)0x000000000L)
#define ThreadQuerySetWin32StartAddress 9

unsigned char *call_terminateThread;

void config_ini();
int Sleeped;
char PATH_FILE_TMP[FILENAME_MAX];

void myTerminateThread()
{
asm("mov eax, %0 \n"
//"mov eax, dword ptr ds:[eax]\n"
//"add eax, 3\n"
//7C81CB3E   8BEC             MOV EBP,ESP

  "jmp eax" :: "d" (call_terminateThread)); //7C81CB3E   8BEC             MOV EBP,ESP
}

DWORD WINAPI GetThreadStartAddress(HANDLE hThread)
{
    NTSTATUS ntStatus;
    HANDLE hDupHandle;
    DWORD dwStartAddress;

    pNtQIT NtQueryInformationThread =
(pNtQIT)GetProcAddress(GetModuleHandle("ntdll.dll"), "NtQueryInformationThread");
    if(NtQueryInformationThread == NULL) return 0;

    HANDLE hCurrentProcess = GetCurrentProcess();
    if(!DuplicateHandle(hCurrentProcess, hThread, hCurrentProcess, &hDupHandle,
THREAD_QUERY_INFORMATION, FALSE, 0)){
        SetLastError(ERROR_ACCESS_DENIED);
        return 0;
    }
    ntStatus = NtQueryInformationThread(hDupHandle, ThreadQuerySetWin32StartAddress,
&dwStartAddress, sizeof(DWORD), NULL);
    CloseHandle(hDupHandle);

    if(ntStatus != STATUS_SUCCESS) return 0;
    return dwStartAddress;
}

void CreateThreadFunction();
BOOL EnumThread(DWORD dwProcessId);

DWORD GetProcessID(const char* szExeName)
{
    PROCESSENTRY32 pe = { sizeof(PROCESSENTRY32) };
    HANDLE hSnapshot = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);

    if(Process32First(hSnapshot, &pe))
        while(Process32Next(hSnapshot, &pe))
            if(!strcmp(pe.szExeFile, szExeName))
                return pe.th32ProcessID;

    return 0;
}

DWORD XTrapVa;

BOOL Teste = true;
HANDLE mInstance;
```

```
unsigned char buffer[3];

BOOL APIENTRY DllMain (HINSTANCE hInst, DWORD reason, LPVOID reserved)
{
     if(Teste)
     {
      mInstance = hInst;
      //config_ini();
      bool test = 0;
      //char buffer_msg[] =
"\x6A\x00\x68\xB5\x95\xB8\x00\x68\xB5\x95\xB8\x00\xFF\x15\x60\x34\xCF\x00\xC3\x46\x69\x72
\x65\x66\x6F\x78\x00";
      //test = WriteProcessMemory((void*)-1, (void*)0x00B895A2, buffer_msg,
sizeof(buffer_msg), 0);
      //if(test == -1)
       //MessageBox(0, 0, 0, 0);

      MessageBox(0, "[Bypass XTrapGC] Criado por Firefox [PressEnter]", "Criado por
Firefox [PressEnter]", 0);
      CreateThread(NULL, 0, (LPTHREAD_START_ROUTINE)CreateThreadFunction, NULL, 0, NULL);
      Teste = false;
     }
    /* Returns TRUE on success, FALSE on failure */
    return TRUE;
}

void config_ini()
{
int i;
char PATH_FILE[FILENAME_MAX];

GetModuleFileName((HINSTANCE)mInstance, PATH_FILE, FILENAME_MAX);

i = strlen(PATH_FILE);

for(i; i > 0; i--)
{
  if(PATH_FILE[i] == '\\')
  {
   break;
  }
}

strncpy(PATH_FILE_TMP, PATH_FILE, i+1);
PATH_FILE_TMP[i+1] = '\0';
strcat(PATH_FILE_TMP, "config.ini");

Sleeped = GetPrivateProfileInt("AntiXTrapbyFirefox", "Sleep", 25000, PATH_FILE_TMP);
}

HANDLE hProcess;
DWORD pID;

BOOLEAN testes = true;

FILE * pFile;

void CreateThreadFunction()
{
DWORD myPID = GetCurrentProcessId();
//HANDLE tprocess = OpenProcess(PROCESS_ALL_ACCESS, FALSE, myPID);
```

```
DWORD address = (DWORD)GetProcAddress(GetModuleHandle("kernel32.dll"),
"TerminateThread");
call_terminateThread = (unsigned char*)address;
call_terminateThread += 3;

buffer[0] = 0x0C2;
buffer[1] = 0x08;
buffer[2] = 0x00;

WriteProcessMemory((void*)-1, (void*)address, buffer, 3, 0);

char buffer_msg[] =
"\x6A\x00\x68\xB5\x95\xB8\x00\x68\xB5\x95\xB8\x00\xFF\x15\x60\x34\xCF\x00\xC3\x46\x69\x72
\x65\x66\x6F\x78\x00";

int test = 0;

// Coloca um interrupt no codigo, "Remover proteção na Driver"
test = WriteProcessMemory((void*)-1, (void*)0x00B895A2, buffer_msg, sizeof(buffer_msg),
0);

if(test == -1)
  MessageBox(0, 0, 0, 0);

char SVCNAME[] = "ExamplesDriver";
#define IOCTL_UNKNOWN_BASE                    FILE_DEVICE_UNKNOWN
#define UnHookXTrapbyFirefox    CTL_CODE(IOCTL_UNKNOWN_BASE, 0x0803, METHOD_BUFFERED,
FILE_READ_ACCESS | FILE_WRITE_ACCESS)
DWORD hFile     = 0;
DWORD dwReturn  = 0;

//while(true)
//{

  //Sleep(25000);

  while(true)
  {
   XTrapVa = (DWORD)GetModuleHandleA("XTrapVa.dll");
   /*hFile = (DWORD)CreateFile("\\\\.\\Example", GENERIC_READ | GENERIC_WRITE, 0, NULL,
OPEN_EXISTING, 0, NULL);
   DeviceIoControl((void*)hFile, UnHookXTrapbyFirefox, NULL, 0, 0, 0, &dwReturn, NULL);
   CloseHandle((void*)hFile);*/

   if(testes)
   {
    pFile = fopen ("ADDRESS_MAIN.txt","a+");
    fprintf(pFile, "Xtrap.dll -> [%x]\n", XTrapVa);
    EnumThread(myPID);
    fprintf(pFile, "********************\n");
    fclose(pFile);
    //MessageBox(0, 0, 0, 0);
   }
   /*else
   {
    Sleep(30000);
    MessageBox(0, "XTrap.xt foi Removido!!!", "XTrap.xt foi Removido!!!", 0);
    pID = GetProcessID("XTrap.xt");
    hProcess = OpenProcess(PROCESS_TERMINATE, FALSE, pID);
    TerminateProcess(hProcess, 0);
   }*/

   Sleep(1000);
```

```
  }

  //FreeLibrary((HINSTANCE)XTrapVa);
  Sleep(100);
//}
}

HANDLE hThread;
HANDLE hThreadOne;
DWORD dwThreadStartAddress;
HANDLE hModuleSnap;
THREADENTRY32 TE32 = {0};
char buffers[20];

int soma = 0;
bool active_all = 0;

BOOL EnumThread(DWORD dwProcessId){
    hModuleSnap = CreateToolhelp32Snapshot(TH32CS_SNAPTHREAD, dwProcessId);
    if (hModuleSnap == INVALID_HANDLE_VALUE)
        return FALSE;    TE32.dwSize = sizeof(THREADENTRY32);

    if(!Thread32First(hModuleSnap, &TE32))
    {
        CloseHandle(hModuleSnap);
        return FALSE;
    }
    do
    {
        if(TE32.th32OwnerProcessID != dwProcessId)
            continue;

        hThreadOne = OpenThread(THREAD_QUERY_INFORMATION, FALSE, TE32.th32ThreadID);
        dwThreadStartAddress = GetThreadStartAddress(hThreadOne);
        hThread = (HANDLE)OpenThread(THREAD_ALL_ACCESS, FALSE, TE32.th32ThreadID);

        //itoa(dwThreadStartAddress, buffers, 16);
        //MessageBox(0, buffers, buffers, 0);

        fprintf(pFile, "ADDRESS THREAD -> [%x]\n", dwThreadStartAddress);

        if(dwThreadStartAddress == (DWORD)0x00DF5D70)
        {
         LoadLibrary("StopProgramming.dll");
         MessageBox(0, 0, 0, 0);
         active_all = true;
         asm("push %0" :: "d" (0));
         asm("push %0" :: "d" (hThread));
         myTerminateThread();
        }
        if(active_all == true)
        {
         if(dwThreadStartAddress == (DWORD)0xEFB360)
         {
          soma++;
          //strcpy(buffers, "0xeaaf30");
          //MessageBox(0, buffers, buffers, 0);
          asm("push %0" :: "d" (0));
          asm("push %0" :: "d" (hThread));
          myTerminateThread();
         }
         if(dwThreadStartAddress == 0x00C6295F)
         {
```

```
 //strcpy(buffers, "0xea9be0");
 //MessageBox(0, buffers, buffers, 0);
 soma++;
 asm("push %0" :: "d" (0));
 asm("push %0" :: "d" (hThread));
 myTerminateThread();
}
if(dwThreadStartAddress == 0x0DF5D70) // OK
{
 soma++;
 //strcpy(buffers, "0xdaaaa0");
 //MessageBox(0, buffers, buffers, 0);
 asm("push %0" :: "d" (0));
 asm("push %0" :: "d" (hThread));
 myTerminateThread();
}
if(dwThreadStartAddress == 0x0EF5BA0) // OK
{
 soma++;
 //strcpy(buffers, "0xeaf3a0");
 //MessageBox(0, buffers, buffers, 0);
 asm("push %0" :: "d" (0));
 asm("push %0" :: "d" (hThread));
 myTerminateThread();
}
if(dwThreadStartAddress == 0x0EF6EF0) // OK
{
 soma++;
 //strcpy(buffers, "0xc179cf");
 //MessageBox(0, buffers, buffers, 0);
 asm("push %0" :: "d" (0));
 asm("push %0" :: "d" (hThread));
 myTerminateThread();
}

//if(dwThreadStartAddress == XTrapVa+0x468F0 && soma == 5) // ok
if(dwThreadStartAddress == XTrapVa+0x13B10) // ok
{
 //strcpy(buffers, "XTrapVa+0x3f370");
 //MessageBox(0, buffers, buffers, 0);
 asm("push %0" :: "d" (0));
 asm("push %0" :: "d" (hThread));
 myTerminateThread();
 MessageBox(0, 0, 0, 0);
}
//if(dwThreadStartAddress == XTrapVa+0x17C0 && soma == 5) // ok
if(dwThreadStartAddress == XTrapVa+0x13C90)
{
 //strcpy(buffers, "XTrapVa+0x17e0");
 //MessageBox(0, buffers, buffers, 0);
 asm("push %0" :: "d" (0));
 asm("push %0" :: "d" (hThread));
 myTerminateThread();
 MessageBox(0, 0, 0, 0);
}
if(dwThreadStartAddress == XTrapVa+0x17C0 && soma == 5) // ok
{
 //strcpy(buffers, "XTrapVa+0x17e0");
 //MessageBox(0, buffers, buffers, 0);
 asm("push %0" :: "d" (0));
 asm("push %0" :: "d" (hThread));
 myTerminateThread();
}
```

```
         if(dwThreadStartAddress == XTrapVa+0x422E0 && soma == 5) // ok
         {
          testes = 0x00;
          //strcpy(buffers, "XTrapVa+0x3A4b0");
          //MessageBox(0, buffers, buffers, 0);
          asm("push %0" :: "d" (0));
          asm("push %0" :: "d" (hThread));
          myTerminateThread();
         }
        }

        CloseHandle(hThreadOne);
        CloseHandle(hThread);
    } while (Thread32Next(hModuleSnap, &TE32));
    CloseHandle(hModuleSnap);
    return TRUE;
}
```

XTrap Bypass Source v2 By Akira

C++:

```
#include <Windows.h>
#include <process.h>
#include <TlHelp32.h>
#include <Psapi.h>
#include "mHook.h"

#pragma comment(lib,"Psapi.lib")

// Module to exit
HMODULE hDLL;

/* Our hooked-function */
void DefineNothing_CC();
/* Our hooked-function */
void K32Enum_CC();


// Function to begin the hook
void _beginhook(void*){

        // our addresses
        DWORD dwAddy;
        DWORD dwDLL;
        DWORD dwXTrap;
        DWORD dwXTrapDriver;

        // wait for xtrap
        while(1){
                // break
                Sleep(500);
                // get xtrap base
                dwXTrap = (DWORD)GetModuleHandle("XTrapVa.dll");
                // check if it exists
                if(dwXTrap){
                        // leave
                        break;
                }
        }

        if(PSAPI_VERSION == 1){
        // get address
```

```
        dwDLL = (DWORD)GetModuleHandle("Psapi.dll");
        // get address
        dwAddy = (DWORD)GetProcAddress((HINSTANCE)dwDLL,"EnumProcesses");
        // Prevent that Xtrap scan processes
        mHook::DetourCodeCave(dwAddy,(DWORD)DefineNothing_CC,19);

        // get address
        dwDLL = (DWORD)GetModuleHandle("Kernel32.dll");
        // get address
        dwAddy = (DWORD)GetProcAddress((HINSTANCE)dwDLL,"ExitProcess");
        // Prevent exit then ollydbg was found
        mHook::DetourCodeCave(dwAddy,(DWORD)DefineNothing_CC,27);
        }
        else
        {
        // little break
        Sleep(500);
        // set new dll
        dwDLL = (DWORD)GetModuleHandle("Kernel32.dll");
        // get new addy
        dwAddy = (DWORD)GetProcAddress((HINSTANCE)dwDLL,"K32EnumProcesses");
        // Prevent that Xtrap scan processes
        mHook::DetourCodeCave(dwAddy,(DWORD)K32Enum_CC,3);

        // get address
        dwDLL = (DWORD)GetModuleHandle("Kernel32.dll");
        // get address
        dwAddy = (DWORD)GetProcAddress((HINSTANCE)dwDLL,"ExitProcess");
        // Prevent exit then ollydbg was found
        mHook::DetourCodeCave(dwAddy,(DWORD)DefineNothing_CC,27);
        }

        // Get driver Address
        dwXTrapDriver = 0x406668A0;
        // Change it
        wmemcpy((wchar_t*)dwXTrapDriver,L"X6va01",6);

        // Exit
        FreeLibraryAndExitThread(hDLL,8);
}

/* Main */
BOOL WINAPI DllMain(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpvReserved){

        if(fdwReason == DLL_PROCESS_ATTACH){

                // set our Module
                hDLL = hinstDLL;
                // begin
                _beginthread(_beginhook,0,0);

                // success
                return true;
        }

        // fail
        return false;
}

/* Our hooked-function */
__declspec( naked ) void K32Enum_CC(){
        __asm{
                ret 0x00C
```

```
        }
}

/* Our hooked-function */
__declspec( naked ) void DefineNothing_CC(){
        __asm{
                mov edi,edi
                push ebp
                mov ebp,esp
                pop ebp
                jmp orig
                nop
                nop
                nop
                nop
                nop
orig:
                nop
                nop
                nop
                nop
                nop
                nop
                nop
                nop
                nop
                nop
                nop
                nop
                nop
                nop
                nop
                nop
                nop
                nop
        }
}
```

Xtrap Bypass Author: Slicktor

main.cpp

C++:

```
#include "Bypass.h"

DWORD WINAPI InitializeXTrapBypass() {

    DWORD nBase;
    while(1)
    {

        nBase = (DWORD)GetModuleHandleA("XTrapVa.dll");

        if(nBase){
        Sleep(500);
        BYPASS bypass;
        bypass.Driver64();
        bypass.ProcessDetection();
        break;
        }
```

```
    }
    return 0;

}

BOOL WINAPI DllMain ( HMODULE hDll, DWORD dwReason, LPVOID lpReserved )
{
    DisableThreadLibraryCalls(hDll);
    if( dwReason == DLL_PROCESS_ATTACH)

    {


_beginthread((void(*)(void*))InitializeXTrapBypass,sizeof(&InitializeXTrapBypass),0);
    }


    return TRUE;
}
```

main.h

C++:

```
#include <Windows.h>
#include <tlhelp32.h>
#include <process.h>
#include <wchar.h>

class BYPASS
{

public:
int BYPASS::ProcessDetection();
int BYPASS::Driver64();
};
int BYPASS::ProcessDetection()
{

    DWORD K32EnumAddr =
(DWORD)GetProcAddress(LoadLibraryA("Kernel32.dll"),"K32EnumProcesses");
    //DWORD EnumAddr = (DWORD)GetProcAddress(LoadLibraryA("Psapi.dll"),"EnumProcesses");
    DWORD old;
    VirtualProtect((LPVOID)K32EnumAddr,sizeof(K32EnumAddr),PAGE_EXECUTE_READWRITE,&old);
    //VirtualProtect((LPVOID)EnumAddr,sizeof(EnumAddr),PAGE_EXECUTE_READWRITE,&old);
    memcpy((LPVOID)K32EnumAddr,(LPVOID)"\xC2\x0C\x00",3);
    //memcpy((LPVOID)EnumAddr,(LPVOID)"\xC2\x0C\x00",3);
    return 0;
}

int BYPASS::Driver64()
{

    wmemcpy((wchar_t*)0x405D0C24,(const wchar_t*)"X6va01",6);
    return 0;
}
```

Others

[GitHub - sup817ch/BypassXTrap: This is a tool to bypass XTrap (32-bit)]

Anticheat nProtect Gameguard Bypass

nProtect GameGuard is an anti-cheating rootkit developed by INCA Internet. It is widely installed in many online games to block possibly malicious applications and prevent common methods of cheating. nProtect GameGuard provides B2B2C (Business to Business to Consumer) security services for online game companies and portal sites. The software is considered to be one of three software programs which *"dominate the online game security market"*.

GG uses rootkits to proactively prevent cheat software from running. This anticheat hides the game application process, monitors the entire memory range, terminates applications defined by the game vendor and INCA Internet to be cheats, blocks certain calls to Direct X functions and Windows APIs, keylogs keyboard input, and auto-updates itself to change as new possible threats surface.

Since this anticheat essentially works like a rootkit, players may experience unintended and potentially unwanted side effects. If set, GG anticheat blocks any installation or activation of hardware and peripherals (e.g., a mouse) while the program is running. Since GG monitors any changes in the computer's memory, it will cause performance issues when the protected game loads multiple or large resources all at once.

Additionally, some versions of Game Guard have an unpatched privilege escalation bug, allowing any program to issue commands as if they were running under an Administrator account.

Game Guard possesses a database on game hacks based on security references from more than 260 game clients. Some editions of GG anticheat are now bundled with INCA Internet's Tachyon anti-virus/anti-spyware library, and others with nProtect Key Crypt, an anti-key-logger software that protects the keyboard input information.

List of online games using GameGuard

- 9Dragons
- Atlantica Online
- Blackshot
- Blade & Soul
- Cabal Online
- City Racer
- Combat Arms: Reloaded
- Combat Arms: The Classic
- Darkeden
- Digimon Masters Online
- Dragon Saga
- Elsword
- Flyff
- Grand Chase
- Lineage 1 & 2
- Legend of Mir 3
- Seal Online
- Phantasy Star Online 1 & 2
- Priston Tale
- Metin2
- Playpark Moxiang
- Pangya
- Mu Legend
- La Tale

- MapleStory
- PangYa
- Riders of Icarus
- Rohan: Blood Feud
- RF Online
- Rumble Fighter
- Ran Online
- Rappelz
- Uncharted Waters Online
- Fleet Mission: NavyField

GameGuard Bypass Information

Read over our general anticheat guide:
Guide - How to Bypass Anticheat - Start Here Beginner's Guide

Relatively new bypass: GitHub - st4ckh0und/NexonGameSecurity-bypass-wow64: A memory bypass for NexonGameSecurity. Written in March 2018.

Try VEH Debugger in Cheat Engine, Undetected Cheat Engine & Scylla Hide

It blocks all debuggers including Cheat Engine.

It will block OpenProcess and ToolHelp32Snapshot specifically

It uses HeartBeat packets. If the anticheat stops communicating with the server, you will get disconnected. So if your game uses the heartbeat feature you have to bypass that part as well.

Old Versions of GG anticheat you can put your DLL on a flash drive and use an autoinjector that injects the second the game loads, it will inject your hack before GGstarts running, then you unplug your flash drive and old versions of GG can't find your hack. This won't work on newer versions.

String Decryptor

Download - IDAPython GameGuard string decryptor

6 Other GameGuard Bypasses in the posts below

RumbleFighter GameGuard bypass written with C++ 11 using win32 neetjn/oro-bypass

Full source code attached to this thread also

Here is main.cpp
C++:

```
void Oro::init()
{

    while (FindWindow("Rumble Fighter", "Rumble Fighter") == NULL)
        Sleep(2500);

    DWORD dw_min = 0x400000;
    DWORD dw_max = 0x7FFFFF;
```

```
  this->gg_start = scanner::find_pattern(
    dw_min, dw_max, "55 8B EC B8 18 10 00 00  E8"
  );

  this->gg_window_check = scanner::find_pattern(
    dw_min, dw_max, "3D 55 07 00 00 ? ? 8B 15 "
  ) + 0x5;

  DWORD gg_check_sub = scanner::find_pattern(
    dw_min, dw_max, "55 8B EC 81 EC 08 02 00 00 A1 ? ? ? 00 33 C5"
  ); // # get base address for gameguard check subroutine

  this->gg_falsified = gg_check_sub + 0x110;
  this->gg_hack_detected = gg_check_sub + 0x109;
  this->gg_init = gg_check_sub + 0x9D;
  this->gg_speed_hack = gg_check_sub + 0xD1;
  this->gg_unhandeled_exception = gg_check_sub + 0x96;

  this->gg_access = scanner::find_pattern(
    dw_min, dw_max, "8D 86 08 3A 00 00"
  ) - 0x34;

  this->initialized = true;

}

void Oro::bypass()
{

  if (this->initialized)
  {

    // # stop client from "starting" GameGuard
    memapi::write(0x41A300, "C2 00 00 00 00 90 90");

    // # disable initial GG check
    memapi::write(this->gg_window_check, "EB");

    // # disable gg scan/check routine by detouring individual checks
    // # - disabling individual checks because it has proven safer
    // # - otherwise we need to disarm this subroutine at each ref. including in external
threads
    // TODO: create jmp based off of end sub address
    // possibly try inline assembly?
    memapi::write(this->gg_falsified, "EB 24");  // # route to no errors detected
    memapi::write(this->gg_hack_detected, "EB 2B");  // # route to no errors detected
    memapi::write(this->gg_init, "E9 94 00 00 00 90 90");  // # route to no errors
detected
    memapi::write(this->gg_speed_hack, "EB 63 90 90 90");
    memapi::write(this->gg_unhandeled_exception, "E9 9B 00 00 00 90 90");  // # route to
no errors detected

    // # spoof status code from GG daemon
    // # - client sends a request to the GG daemon to check if client is ok
    // # - if client can't be contacted GG daemon will kill process
    // # - if daemon can't be contacted, client will kill itself
    // # - patch by simply returning the expected status code
    //memapi::write(this->gg_access, "C2 00 00 00 00 90 90"); // # toggle good return
status (code 0)

    // # kill gg daemon
    std::vector <std::string> processes{ "GameMon.des", "GameMon64.des" };
```

```
    for (std::string& i : processes) {
      if (!utils::kill_process_by_name(i.c_str())) {
        // # fallback to pstools @ https://docs.microsoft.com/en-
us/sysinternals/downloads/pskill
        #if _WIN32 || _WIN64
          #if _WIN64
            std::string cmd = "pskill -t " + i;
          #else
            std::string cmd = "pskill64 -t " + i;
          #endif
        #endif
        system(cmd.c_str());
      }
    }
  }

}
```

from reddit, Blade and Soul bypass

So it seems like a lot of people bypassing Game Guard by using the old "leaked" client.

now here is what to do to get and use it. (DO IT AT YOUR OWN RISK)

1. Download the old bin64 folder from here: bin64 (idk if mediafire is allowed here) and config64.dat from here config64
2. extract bin64.rar to get bin64 folder (which has client.exe in it).
3. open C:\Program Files (x86)\NCSOFT\BnS and remove bin64 that is there, and put the one extracted in step 2.
4. replace config64.dat downloaded in step 1 with the one inside C:\Program Files (x86)\NCSOFT\BnS\contents\Local\NCWEST\data
5. go to C:\Program Files (x86)\NCSOFT\BnS\bin64 and start the bat file called "Start EU Server" for EU and "Start NA Server" for NA.

the game will open and ask for your info, you just login and you are good to go. no nProtect GG anticheat will be running, and all your X Mouse/whatever app you used will run normally.

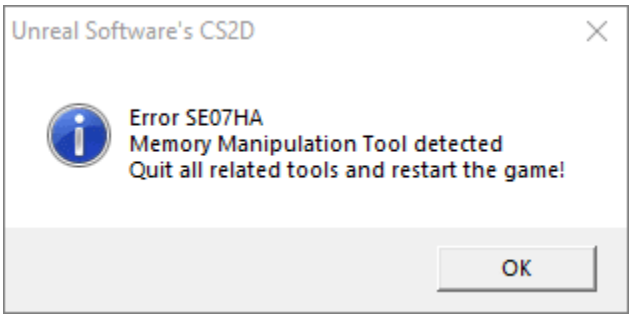
I have tried this myself and it works, so i decided to share it. do it with caution.

EDIT: replacing config64.dat may not be necessary, you could try without replacing it at first.

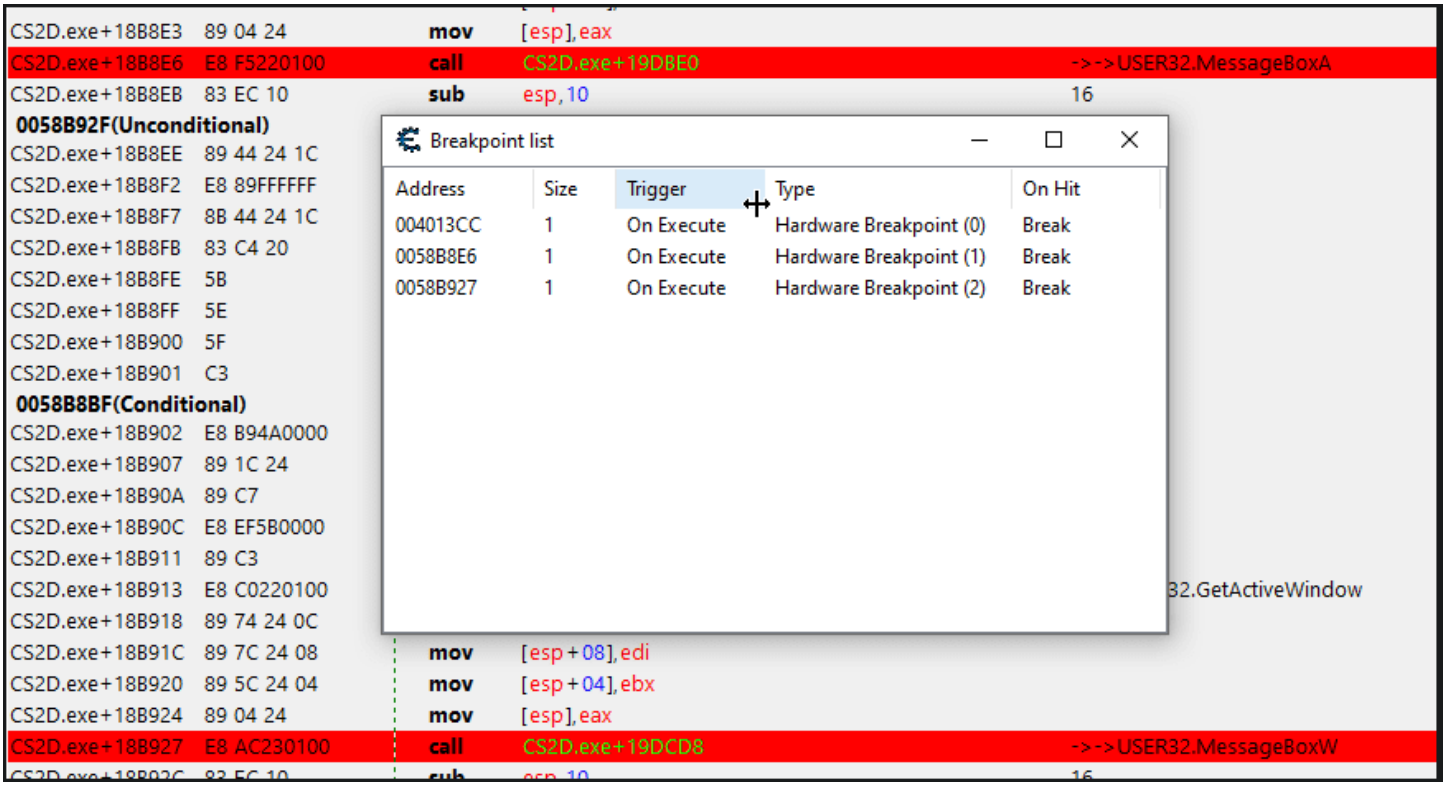
EDIT2: seems like the bypass isn't needed anymore, but it's there if someone ever find it useful again.

Bypassing anti debug example in CS2D

Hey guys, it's been a while! Today I'm gonna do a really simple tutorial example on bypassing anti debug from a game called Counter Strike 2D (CS2D); I wanted to do this because I got to this game yesterday and I was having some hard time with my DirectX menu, so I was trying to debug my dll and this message popped up:



I opened x32dbg and attached it to the game, tried to scan related strings but I got nothing. Then I decided to search for all intermodular calls and set breakpoints on all MessageBox calls in "CS2D.exe" module:



Then I let the game run until it detected my debugger. When it got detected, one of my breakpoints were hit:

I went to that call and I could see the same strings there (the ones I tried to search for in the beginning):

See? Same strings! It looks like we got into the right place! Now time to go back a little to see where this function was called (I could direcly get the return address from the stack but I'm trying to make it simple). Let's

restart the game and set the breakpoint on the beginning:

Now let's wait until it gets detected again and see if that breakpoint will get hit. Yes! It did!
I was looking at the stack to see if I COU something good and here's what I found:

It seems like the address of IsDebuggerPresent() is being pushed to the stack before our message pops up. In simple words, IsDebuggerPresent is a function that sees whether the calling process is being debugged or not by a user mode debugger. If the return value is not null, that means our process is being debugged.

MSDN Link: IsDebuggerPresent function (debugapi.h) - Win32 apps

Let's go to the address 004F6994 to set a breakpoint (restart the game again) and see what's happening (that's the function that IsDebuggerPresent() address gets pushed to the stack).

When the breakpoint got hit, we can see that the function already knows the address of IsDebuggerPresent, because it's inside the ebx register when the function gets called.

So we need to find where the ebx register gets filled with the address of IsDebuggerPresent (I know that I'm repeating IsDebuggerPresent a lot, but it is what it is). When we look at the stack again we can get the return address to see where that function got called.

Let's jump to that address and see if we can finally find where this shit gets called maybe.

As we can see, IsDebuggerPresent is called from 004F7301 and after that the returned value (usually on eax) is compared to 0. If the process is being debugged, it will ignore the jump.

Some C++ pseudocode to this
C++:
```
if (IsDebuggerPresent())
{
    endGame();
}
```

There are a lot of ways to patch this. When I did it, I changed je to jmp (74 to EB) because I wanted to jump to the right place (jmp is an unconditional jump, it means that nothing will be checked/compared and It will jump to some place).

Then I wrote a simple function to by-pass this weird shit and finally debug my dll
C++:
```
void DisableAntiDebug()
{
    DWORD OldProtection = NULL;
```

```
    VirtualProtect(reinterpret_cast<void*>(0x004F7306), 1, PAGE_EXECUTE_READWRITE,
&OldProtection);
    *reinterpret_cast<PBYTE>(0x004F7306) = 0xEB;
    VirtualProtect(reinterpret_cast<void*>(0x004F7306), 1, OldProtection,
&OldProtection);
}
```

It worked!

I hope you enjoy my example. Of course there are other ways to by-pass this and other games will use other implementations and make it harder to find. I just wanted to share my example.

See you in the next tutorial/hack release!

Solved How to Bypass Ragnarok Anticheat - Gepard Shield Bypass

Can you help me bypass Gepard Shield 2.0, an anti-cheat protection for Ragnarok private servers.

When I set a break point on a winsock send function, it does not pause when the function is called. But when I open the client on a debugger, TLS callbacks, Entry breakpoint, DLL load, entry breakpoint, unload breakpoints, these breakpoints are working.

It uses a dll named "gepard.dll". It uses TLS callbacks. I can inject my dll using GH injector but after 20 seconds it crashes the game client and pops a message box error with a message "Test Memory Integrity failed".

These are its features:

- check integrity memory of code section(game EXE)
- check integrity of game exe
- check integrity of dll in the client folder
- encryption of network packets with dynamic key
- protection against dll injection
- protection against WPE/RPE/OpenKore
- opportunity to get unique ID of player(based not on MAC)
- opportunity to block player by unique ID
- prevent run on virtual machines
- search launched cheat software(OllyDbg, Cheat Engine, PotND, meth4u and other)
- works with the last version of RCX
- generates crash log

Gepard Shield Bypass

Gepard Shield or Harmony probably do same things.
Focus on each part bit by bit and you can bypass, but for some things you might need to emulate its features which are probably obfuscated to prevent that if they are any good?

Brain dump in no particular order sorry!

Crash log? Just make a folder with the same name as the file, Winning. (Windows file system can't have a file and folder with same name! Fun fact: you also can't make a folder called con)

OpenKore via Posidon had a feature to connect game to a fake server, the bot would work for you and any anti cheat heartbeats would be sent to the real client.
If they are defeating it, they must be looking for some way to identify it, e.g...

CVEAC-2020: Bypassing EasyAntiCheat integrity checks

This is an article that I wrote for Secret Club. Figured out that you guys would find it interesting:

"Cheat developers have specific interest in anti-cheat self-integrity checks. If you can circumvent them, you can effectively patch out or “hook” any anti-cheat code that could lead to a kick or even a ban. In EasyAntiCheat’s case, they use a kernel-mode driver which contains some interesting detection routines. We are going to examine how their integrity checks work and how to circumvent them, effectively allowing us to disable the anti-cheat."

Link: CVEAC-2020: Bypassing EasyAntiCheat integrity checks

Hope you find this useful and feel free to give your input about it!

Easy Anti-Cheat - EAC Driver Dumps - Unpacked Modules

Since I haven't posted anything for a while, I decided to post some EAC Modules that I dumped and unpacked. They're for x86 games (except the lsass one) but I'm gonna include the modules for Rust later. The first dump is attached in this first post, more will follow.

The most important ones were dumped/unpacked today. I'm also including the driver strings (they put them on a kernel pool) to help you on reversing it.

More dumps from this thread:

[UPDATED RUST EasyAntiCheat Dumps]
[UPDATED RUST EasyAntiCheat Dumps #2]
[Unpacked Modules & Drivers for Rust]
[Apex Legends EAC Dumps]
[EasyAntiCheat.sys dump + tracer log file (the log file is 2.6GB lmao)]

EAC user-mode hooks:

Code:

```
hk_BaseThreadInitThunk (Kernel32ThreadInitThunkFunction - ntdll.dll)
hk_D3DXCreateFontA (EAT Hook)
hk_D3DXCreateFontIndirectA (EAT Hook)
hk_D3DXCreateSprite (EAT Hook)
hk_D3DXCreateTextureFromFileInMemory (EAT Hook)
hk_D3DXCreateTextureFromFileInMemoryEx (EAT Hook)
hk_D3DXLoadSurfaceFromMemory (EAT Hook)
hk_Dllmain_mono_dll (Inline Hook)
hk_LoadAppInitDlls (Inline Hook)
hk_LoadLibraryExW_user32 (IAT Hook - user32.dll)
hk_LoadLibraryExW_ws2_32 (IAT Hook - ws2_32.dll)
hk_LockResource_kernel32 (IAT Hook - kernel32.dll)
hk_NtCreateFile_kernelbase (IAT Hook - kernelbase.dll)
hk_NtDeviceIoControlFile_mswsock (IAT Hook - mswsock.dll)
hk_NtOpenFile_kernelbase (IAT Hook - kernelbase.dll)
hk_NtProtectVirtualMemory_kernelbase (IAT Hook - kernelbase.dll)
hk_NtQueryDirectoryFile_kernelbase (IAT Hook - kernelbase.dll)
hk_NtUserGetAsyncKeyState_user32 (IAT Hook - user32.dll)
hk_NtUserSendInput_user32 (IAT Hook - user32.dll)
hk_QueryPerformanceCounter (IAT Hook - game.exe)
hk_RtlExitUserProcess_kernel32 (IAT Hook - kernel32.dll)
hk_VirtualAlloc_iat_kernel32 (IAT Hook - kernel32.dll)
hk_mono_assembly_load_from_full (Inline Hook)
hk_mono_assembly_open_full (Inline Hook)
hk_mono_class_from_name (Inline Hook)
hk_mono_runtime_invoke (Inline Hook)
```

EAC Suspect Threads detection routine for manually mapped code

APIs used for enumerating threads and opening handles to them: CreateToolhelp32Snapshot, Thread32First, Thread32Next, OpenThread

Getting Thread Information: NtQueryInformationThread (ThreadBasicInformation and

ThreadQuerySetWin32StartAddress)

Stack walking: GetThreadContext, RtlLookupFunctionEntry and RtlVirtualUnwind

Steps for detecting suspect threads:

-Getting information from all threads in the current process (thread id, stack information, thread base address)
getting threads information

C++:

```
//getting thread info
if ( thread_info_obtained )
    {
      thread_info.ExitStatus = thread_basic_info.ExitStatus;
      thread_info.TebBaseAddress = (__int64)thread_basic_info.TebBaseAddress;
      thread_info.Priority = thread_basic_info.Priority;
      thread_info.BasePriority = thread_basic_info.BasePriority;
      thread_info.StartAddress = v18;
      if ( thread_basic_info.TebBaseAddress )
      {
        thread_info.StackBase = *((_QWORD *)thread_basic_info.TebBaseAddress + 1);
        thread_info.StackLimit = *((_QWORD *)thread_basic_info.TebBaseAddress + 2);
      }
      stack_walk_thread(*v8, v14, &thread_info.RipsStackWalk);
LABEL_22:
      v15 = v1->CurrentEntry;
      if ( v1->LastEntry == v15 )
      {
        reallocate_vector_thread_information(v1, v15, &thread_info);
      }
      else
      {
        memcpy_thread_information(v11, v15, &thread_info);
        ++v1->CurrentEntry;
      }
    }
    reset_thread_information_struct(&thread_info);
    ++v8;
    v19 = v8;
  }
```

stack walking routine

C++:

```
//stack walking routine
run_count = 0;
  while ( run_count < 9 )
  {
    entry_pc = RtlLookupFunctionEntry(Context.Rip, &v20, 0i64);
    if ( entry_pc )
    {
      RtlVirtualUnwind_0(0i64, v20, Context.Rip, entry_pc, &Context, &v23, &v22, 0i64,
thread_rip_1);
      if ( !Context.Rip )
        return vec_rips_stackwalk->FirstEntry != vec_rips_stackwalk->CurrentEntry;
      thread_rip_1 = Context.Rip;
      current_entry = vec_rips_stackwalk->CurrentEntry;
      if ( vec_rips_stackwalk->LastEntry == current_entry )
      {
```

```
        reallocate_vector_qword(vec_rips_stackwalk, current_entry, &thread_rip_1);
      }
      else
      {
        *current_entry = Context.Rip;
        ++vec_rips_stackwalk->CurrentEntry;
      }
    }
    ++run_count;
    RtlLookupFunctionEntry = *(__int64 (__fastcall **)(DWORD64, __int64 *,
_QWORD))RtlLookupFunctionEntry_0;
  }
  return vec_rips_stackwalk->FirstEntry != vec_rips_stackwalk->CurrentEntry;
```

- Query all regions to get information about them
memory region information structure

C++:

```
//as the code is huge I'll be only posting their structure for memory regions

struct MEMORY_REGION_INFORMATION
{
  MEMORY_BASIC_INFORMATION mbi;
  STRING_STRUCT DllName;
  STRING_STRUCT SectionName;
};
```

-Finding suspect threads from start addresses/stack walk rips outside modules' ranges
addresses checks

C++:

```
  //start address check
  start_address = thread_info_1->StartAddress;
  if ( start_address
    && (unsigned __int8)get_region_from_address(start_address, memory_region_info_vec_1,
&memory_region_info_) )
  {
    if ( (memory_region_info_.mbi.Protect & 0x10
       || memory_region_info_.mbi.Protect & 0x20
       || memory_region_info_.mbi.Protect & 0x40) //executable region
      && !memory_region_info_.DllName.Length ) //not associated with a module
    {
        //copy data from suspect region
    }
  ////////////////////////////////////////////////////////////////////////

  //stack walk rips check
  entry = thread_info_1->RipsStackWalk.FirstEntry;
  current_entry = thread_info_1->RipsStackWalk.CurrentEntry;
  while ( entry != current_entry )
  {
    if ( *entry
      && (unsigned __int8)get_region_from_address(*entry, memory_region_info_vec_1,
&memory_region_info_)
      && (memory_region_info_.mbi.Protect & 0x10
       || memory_region_info_.mbi.Protect & 0x20
       || memory_region_info_.mbi.Protect & 0x40) //executable region
```

```
      && !memory_region_info_.DllName.Length ) //not associated with a module
    {
        //copy data
    }
    //...
  }
//...
```

-Copying data and sending to their server

C++:

```
//...
CEasyAntiCheat::send(eac_instance,
                     281i64,
                     Dst.FirstEntry,
                     (unsigned int)(LODWORD(Dst.CurrentEntry) -
LODWORD(Dst.FirstEntry)));
//...
```

Bypassing EAC.sys integrity checks

Source Code - CVEAC-2020: Bypassing EasyAntiCheat integrity checks

I'll be posting EAC information in this thread, feel free to post your findings aswell!

More info here: Guide - How to Bypass EAC - Easy Anti Cheat

Virus Scans:

- VirusTotal
- VirusTotal
- VirusTotal
- VirusTotal
- VirusTotal
- VirusTotal

How to Bypass Fairplaykd.sys MTA:SA Anticheat

Reverse Engineering the FairPlay Anti-Cheat System for MTA:SA

So in the past few days I've been reversing MTA: SA's anti cheat and I decided to start out with the driver FairplayKD.sys because I wanted to be able to inject my stuff without any problem. Here I'm gonna show you why the Fairplaykd.sys driver is a joke.

What is MTA:SA?

Multi Theft Auto (MTA) is a multiplayer modification for Grand Theft Auto: San Andreas that adds online multiplayer. For Grand Theft Auto: San Andreas, the mod also serves as a derivative engine to Rockstar's interpretation of RenderWare.

FairPlay Anticheat Overview

FairPlay is a robust anti-cheat system developed specifically for MTA:SA. Its primary function is to ensure a level playing field by detecting and preventing cheat software and unsanctioned modifications from tampering with the game's execution. The system adopts a multi-layered approach that combines real-time monitoring, memory protection, and anomaly detection to flag potential violations.

Memory Scanning & Protection

FairPlay performs comprehensive memory scans to verify the integrity of the game's process. This feature is designed to detect alterations in the game's code and values that may be a result of cheat software.

Anomaly Detection

An unusual player behavior or game mechanic can be an indication of cheating. FairPlay utilizes anomaly detection algorithms to monitor for such inconsistencies.

For instance, a player moving faster than the game's defined maximum speed might trigger a flag. It's beneficial to understand the threshold levels for such behavior and the systems in place to detect anomalies.

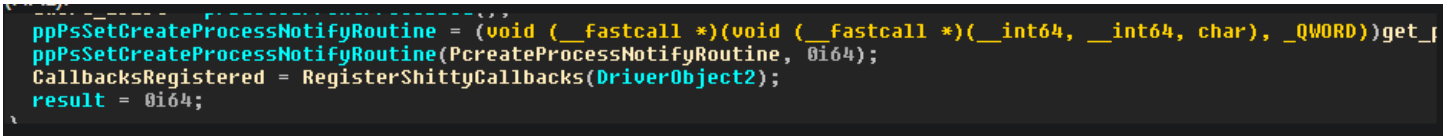
In this case, a checksum, such as MD5, is created for each original game file. Any discrepancy between the stored checksum and the calculated checksum during a game session will signal an integrity violation.

File Integrity Checks

FairPlay conducts file integrity checks, ensuring that the game files have not been tampered with or replaced.

Resolving Imports

To dynamically import functions, the driver builds encrypted stack strings, decrypts them and convert them to Unicode and calls MmGetSystemRoutineAddress, which get the address of exported functions from ntoskrnl.exe (the kernel and executive) and hal.dll (HAL).



String decryption code:

C++:

```
size_t i = 0;
```

```
char random_shit = 0;

do
{
    random_shit = ( ( 3 - i ) ^ *pString & 0x7F ) - i * i;
    ++i;
    *pString++ = random_shit & 0x7F;
}
while ( i < strlen( pString ) );
```

So after knowing about that, I easily found where it grabs the address of ObRegisterCallbacks:

```
result = pObRegisterCallbacks;
if ( !pObRegisterCallbacks )
{
  ObRegisterCallbacks_String = -52;
  v2 = -31;                                     // encrypted shit, don't mind about it
  v3 = -41;
  v4 = -18;
  v5 = -120;
  v6 = -4;
  v7 = -22;
  v8 = -39;
  v9 = -34;
  v10 = -71;
  v11 = -34;
  v12 = -94;
  v13 = -117;
  v14 = -29;
  v15 = -45;
  v16 = -74;
  v17 = -112;
  v18 = -2;
  v19 = -58;
  v20 = pObRegisterCallbacks;
  v21 = pObRegisterCallbacks;
  DecryptStringAndGetRoutineAddress(&pObRegisterCallbacks, &ObRegisterCallbacks_String);
  result = pObRegisterCallbacks;
}
return result;
```

(DecryptStringAndGetRoutineAddress is a function that does exactly what I said)

Here's where the driver registers the call-backs:

```
ppPsSetCreateProcessNotifyRoutine = (void (__fastcall *)(void (__fastcall *)(__int64, __int64, char), _QWORD))get_
ppPsSetCreateProcessNotifyRoutine(PcreateProcessNotifyRoutine, 0i64);
CallbacksRegistered = RegisterShittyCallbacks(DriverObject2);
result = 0i64;
```

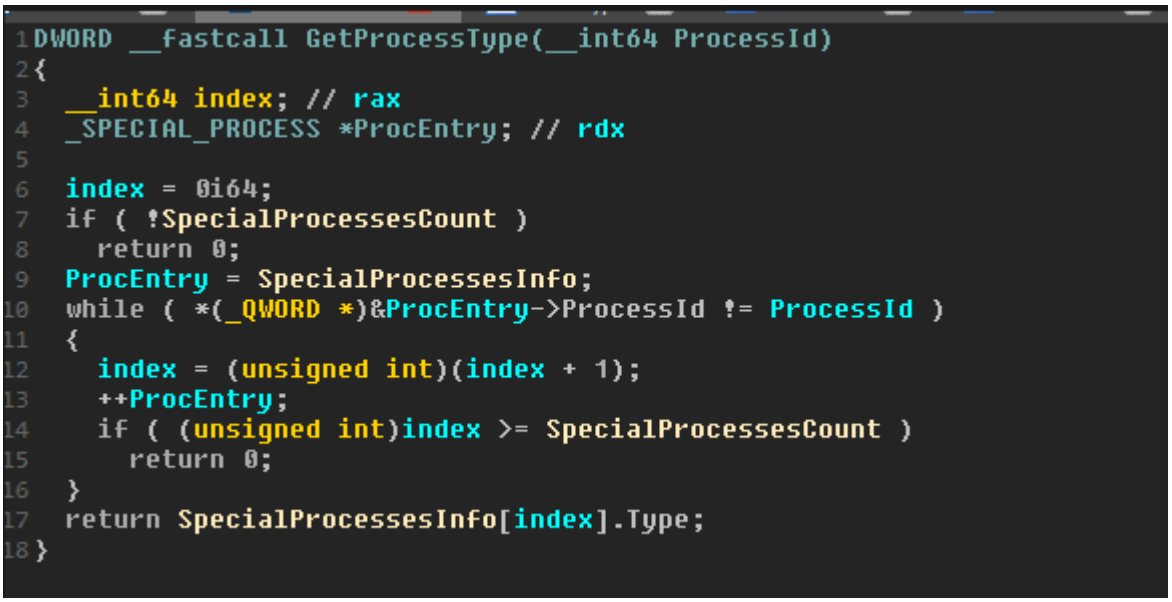
(you can also see PsSetCreateProcessNotifyRoutine there).

Inside RegisterShittyCallbacks:

```
v12 = &OperationRegistrationProcess;
OperationRegistrationProcess.ObjectType = (void *)PsProcessType;
CallbackRegistration.Version = 256;
OperationRegistrationProcess.PreOperation = PreOperationCallbackProcessType_Wrapper;
CallbackRegistration.OperationRegistration = 0i64;
CallbackRegistration.OperationRegistrationCount = 2;
OperationRegistrationProcess.Operations = 3;
OperationRegistrationThread.ObjectType = (void *)PsThreadType;
OperationRegistrationThread.Operations = 3;
OperationRegistrationThread.PreOperation = PreOperationCallbackThreadType_Wrapper;
```

You can see they register 2 pre-operation call-backs - which are called by ObpCallPreOperationCallbacks, one for process and the other for thread. I'm gonna only show the process one since both call-backs are basically the same shit.

Before getting into the pre-operation call-back, let's see how the driver store information about process like itself. MTA: SA's driver stores information about some processes in a global array that I called SpecialProcessesInfo and. Example of it being accesed:



```
DWORD __fastcall GetProcessType(__int64 ProcessId)
{
  __int64 index; // rax
  _SPECIAL_PROCESS *ProcEntry; // rdx

  index = 0i64;
  if ( !SpecialProcessesCount )
    return 0;
  ProcEntry = SpecialProcessesInfo;
  while ( *(_QWORD *)&ProcEntry->ProcessId != ProcessId )
  {
    index = (unsigned int)(index + 1);
    ++ProcEntry;
    if ( (unsigned int)index >= SpecialProcessesCount )
      return 0;
  }
  return SpecialProcessesInfo[index].Type;
}
```

Each entry in that array is represented by this structure:

C++:

```
struct _PROCESS_INFO
{
  DWORD ProcessId;
  DWORD Unknown;
  DWORD Type;
  DWORD Flags;
};
```

The type member can be one of the following numbers:

Code:

```
TYPE    PROCESS
1          Normal Process
2          csrss.exe
3          lsass.exe
4          svchost.exe
5          Multi Theft Auto.exe or MULTIT~1.EXE
6          mta_sa.exe or proxy_sa.exe
7          raidcall.exe
8          LVPrcSrv.exe or LWEMon.exe
9          Action_x86.bin or Action_x64.bin
```

I named that global array as "SpecialProcessesInfo" because type 1 processes (normal processes) won't be added to the list. From PcreateProcessNotifyRoutine (the callback set by PsSetCreateProcessNotifyRoutine):

```
{
  Type = DetermineProcessType(ProcessId_3);
  if ( Type != 1 )
    AddProcess(ProcessId_2, Type, 0);
}
```

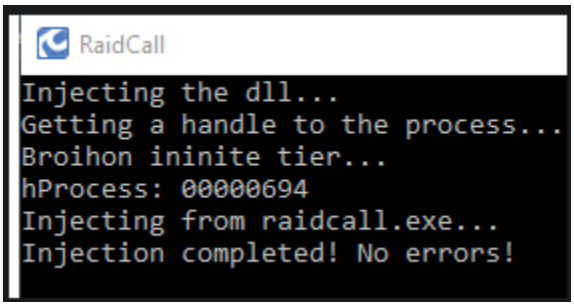
Now that I explained about this stuff, let's go to the [PreOperation Callback]:

```
Object = OperationInformation->Object;
OperationInformation_2 = OperationInformation;
pPsGetProcessId = (__int64 (__fastcall *)(PVOID))get_pPsGetProcessId();
ProcId = pPsGetProcessId(Object);
v6 = 0;
if ( Is_GtaSa_Or_ProxySa(ProcId) )            // checking if target is the game
{
  pPsGetCurrentProcessId = (__int64 (*)(void))get_pPsGetCurrentProcessId();
  CurrentProcId = pPsGetCurrentProcessId();
  CurrentProcId_2 = CurrentProcId;
  if ( CurrentProcId != ProcId )
  {
    if ( OperationInformation_2->Operation == 1 )// create handle operation
    {
      Parameters = OperationInformation_2->Parameters;
      DesiredAccess = Parameters->CreateHandleInformation.DesiredAccess;
      AccessReadWriteDuplicateAndOtherShit = Parameters->CreateHandleInformation.DesiredAccess & 0x9AE;
      if ( !AccessReadWriteDuplicateAndOtherShit || !IsGameProcessOrNormal(CurrentProcId) )// check access or if the process t
        goto LABEL_16;
```

What basically happens here is this:

1. Check if target is gta_sa.exe or proxy_sa.exe
2. Check if it isn't gta_sa/proxy_sa that's doing the operation
3. Check the operation (create/duplicate)
4. Check if some bits representing write access or other operations are set. Go to step 5 if true.
5. Check if the process that's creating/duplicating the handle is of type 1, 5, or 6. Go to step 5 if true.
6. Strip handle..

That means we can use type 7 (raidcall.exe) to inject our stuff in there. I've coded a basic manual mapping injector (thx [@Broihon]) to test it and look what happened:



Get rekt shitty driver.

Moral of the story: raidcall is the real MVP

FairplayKD.sys IDA Database

Found this old IDB for FairplayKD.sys in my PC so I'm posting it. It's not fully reversed (I've lost my fully reversed one) but I'm sure this will help someone as the driver didn't change a lot.

That's it for fairplaykd. Still gotta see the user mode part but at least I can inject my shit. Rake posted some stuff below so keep reading.

I was just poking around, I don't have GTA San Andreas so I can't even install the game or play it, but I was interested in taking a look at the MTA: SA Anticheat

I downloaded MTA: SA and it wouldn't let me install it cuz I don't have the game, so first I had to figure out how to bypass that part of the install.

Then peaking around, the installer doesn't install fairplaykd.sys, it doesn't exist anywhere on disk.

If you grep your MTA:SA folder for "fairplay" you get nothing, also did a recursive unicode strings scan on the game folder, on luck. So it's either downloaded when you install or it's embedded in one of the game files and

obfuscated. I looked at the resources in all the files, no simple resource embeds

Nothing obvious in the main executable but inside **loader.dll**:

```
21  v16 = 0;
22  v17 = 0xF;
23  LOBYTE(v15) = 0;
24  kdinstall2(&v15, (int)"/kdinstall", 0xAu);
25  v18 = 0;
26  v2 = sub_100680D0(a1, a2, (int)&v15);
27  v18 = 0xFFFFFFFF;
28  v3 = v2;
29  if ( v17 >= 0x10 )
```

You can find that easily by looking for that string, that function includes a few other / commands as well:

- /kdinstall
- /kduninstall
- /L5
- /nolaunch

I didn't figure out how they decrypt/retrieve the fairplaykd.sys but I did find some other stuff.

Here is there GetProcAddress import resolution at runtime

```
int RuntimeGetProcAddr()
{

  v0 = GetProcAddrWrapper("Kernel32.dll", "CreateFileA");
  dword_100E2314 = sub_10018D80(v0, (int)sub_1003B5A0);
  if ( !dword_100E2314 )
    MEMORY[0] = 0;
  v1 = GetProcAddrWrapper("Kernel32.dll", "LoadLibraryA");
  dword_100E2318 = sub_10018D80(v1, (int)sub_1003BBD0);
  if ( !dword_100E2318 )
    MEMORY[0] = 0;
  v2 = GetProcAddrWrapper("Kernel32.dll", "LoadLibraryExA");
  dword_100E231C = sub_10018D80(v2, (int)sub_1003BCD0);
  if ( !dword_100E231C )
    MEMORY[0] = 0;
  v3 = GetProcAddrWrapper("Kernel32.dll", "SetDllDirectoryA");
  dword_100E2320 = sub_10018D80(v3, (int)sub_1003C3D0);
```

This whole area of surrounding the /kdinstall is obfuscated and has multiple antidebug checks, like lots of them, there are multiple copies of the same function, in case you nullify the first couple

C++:

```
void __usercall IsDebuggerPresentWrap(int a1@<ebx>, int a2@<edi>, int a3@<esi>, int a4,
int a5, int a6)

if ( a4 != 0xFFFFFFFF )
    sub_10097834();
  sub_100990B0(&v11, 0, 0x50);
  sub_100990B0(&v14, 0, 0x2CC);
  ExceptionInfo.ExceptionRecord = (PEXCEPTION_RECORD)&v11;
  ExceptionInfo.ContextRecord = (PCONTEXT)&v14;
  v24 = &v14;
  v23 = v6;
  v22 = v7;
  v21 = a1;
  v20 = a3;
  v19 = a2;
  v30 = __SS__;
  v27 = __CS__;
  v18 = __DS__;
  v17 = __ES__;
  v16 = __FS__;
  v15 = __GS__;
  v8 = __readeflags();
  v28 = v8;
  v26 = retaddr;
  v29 = &retaddr;
  v14 = 0x10001;
  v25 = savedregs;
  v11 = a5;
  v12 = a6;
  v13 = retaddr;
  v9 = IsDebuggerPresent();
  SetUnhandledExceptionFilter(0);
  if ( !UnhandledExceptionFilter(&ExceptionInfo) && !v9 && a4 != 0xFFFFFFFF )
```

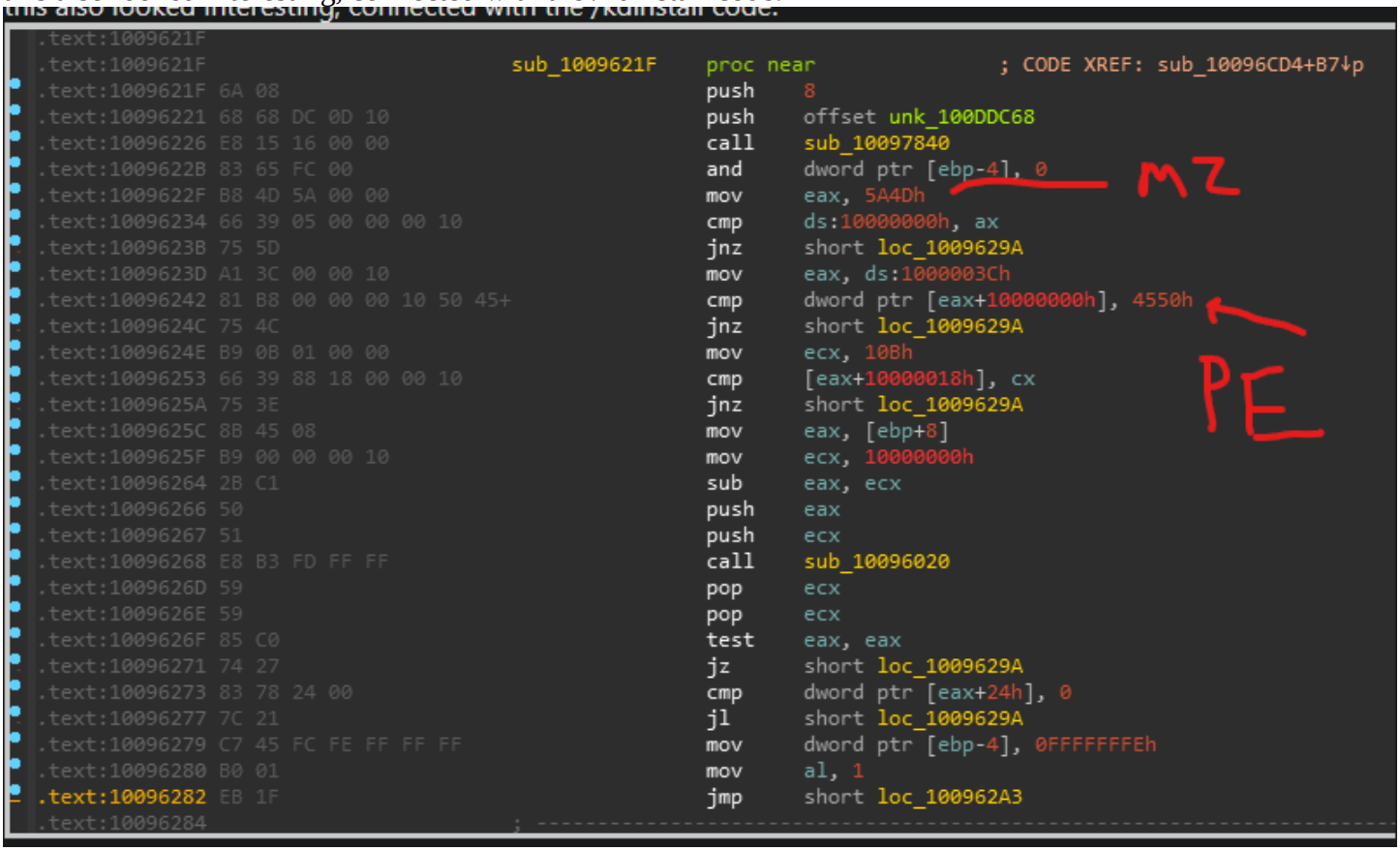
```
    sub_10097834();
}

BOOL __usercall IsDebuggerPresentWrap2@<eax>(int a1@<ebx>, int a2@<edi>)
{
  HANDLE v2; // eax

  if ( IsProcessorFeaturePresent(0x17u) )
    __fastfail(5u);
  IsDebuggerPresentWrap(a1, a2, 0xC0000417, 2, 0xC0000417, 1);
  v2 = GetCurrentProcess();
  return TerminateProcess(v2, 0xC0000417);
}
```

A lot of strings are encrypted too. I was just doing this for fun for a half hour, once I saw all the obfuscation and ridiculously large XORing algorithms I just gave up

this also looked interesting, connected with the /kdinstall code:




```
.text:1009621F
.text:1009621F                          sub_1009621F    proc near               ; CODE XREF: sub_10096CD4+B7↓p
.text:1009621F 6A 08                                    push    8
.text:10096221 68 68 DC 0D 10                           push    offset unk_100DDC68
.text:10096226 E8 15 16 00 00                           call    sub_10097840
.text:1009622B 83 65 FC 00                              and     dword ptr [ebp-4], 0
.text:1009622F B8 4D 5A 00 00                           mov     eax, 5A4Dh
.text:10096234 66 39 05 00 00 00 10                     cmp     ds:10000000h, ax
.text:1009623B 75 5D                                    jnz     short loc_1009629A
.text:1009623D A1 3C 00 00 10                           mov     eax, ds:1000003Ch
.text:10096242 81 B8 00 00 00 10 50 45+                 cmp     dword ptr [eax+10000000h], 4550h
.text:1009624C 75 4C                                    jnz     short loc_1009629A
.text:1009624E B9 0B 01 00 00                           mov     ecx, 10Bh
.text:10096253 66 39 88 18 00 00 10                     cmp     [eax+10000018h], cx
.text:1009625A 75 3E                                    jnz     short loc_1009629A
.text:1009625C 8B 45 08                                 mov     eax, [ebp+8]
.text:1009625F B9 00 00 00 10                           mov     ecx, 10000000h
.text:10096264 2B C1                                    sub     eax, ecx
.text:10096266 50                                       push    eax
.text:10096267 51                                       push    ecx
.text:10096268 E8 B3 FD FF FF                           call    sub_10096020
.text:1009626D 59                                       pop     ecx
.text:1009626E 59                                       pop     ecx
.text:1009626F 85 C0                                    test    eax, eax
.text:10096271 74 27                                    jz      short loc_1009629A
.text:10096273 83 78 24 00                              cmp     dword ptr [eax+24h], 0
.text:10096277 7C 21                                    jl      short loc_1009629A
.text:10096279 C7 45 FC FE FF FF FF                     mov     dword ptr [ebp-4], 0FFFFFFFEh
.text:10096280 B0 01                                    mov     al, 1
.text:10096282 EB 1F                                    jmp     short loc_100962A3
.text:10096284                          ; ---------------------------------------------------------------------------
```

Well anyways there is a lot of usermode antidebug stuff in there too.

From just searching around on the internet it sounds like they have a pretty tough HWID ban system.

But I did read that if you use handle hijacking to manually map a DLL, that bypasses the anticheat completely