

Memory Hacking

게임해킹 분석과 해킹툴 제작

박 시율

목차

1. 개요

1-1. 게임해킹의 종류와 기초

2. 선행지식

3. 메모리해킹 주소 분석

3-1. 원하는 값을 찾는 법

3-2. 변수

3-3. 기능추가에 대해

4. 해킹툴 개발

5. 끝맺음

6. 전체소스

1. 개요

메모리 해킹은 이전부터 많은 곳에 사용되고 있으며 특히 게임분야에 사용되어 사용자들에게 피해를 입히는 기술로 사용됩니다. 또는 프로그램에 기능을 추가하거나 삭제하는 등의 용도로도 사용됩니다.

1-1. 게임해킹의 종류와 기초

해킹프로그램을 칭하는 단어로는 CHEAT, HACK 이 있으며, 흔히 핵이라는 단어로 불립니다. 월핵, 맵핵, 스피드핵, 에임봇 등이 포함됩니다. 핵 개발에 사용되는 언어로는 C++, Delphi, C# 이 주로 사용되며, 매크로의 경우 AutoHotkey가 있습니다.

2. 선행지식

- Assembly
- C++
- Reverse Engineering
- Win32 API

1. Assembly

- 타겟 프로그램을 분석할 때 Assembly로 보여지기 때문에 해당 지식이 필요합니다.

2. C++

- 강력하고 빠르며 본 글에서는 C++을 사용할 것입니다.
Delphi의 경우 C# 이상의 GUI 컴포넌트 지원도 가능하며, C++과 속도 또한 비슷한 수준으로 많이 이용됩니다.

3. ReverseEngineering

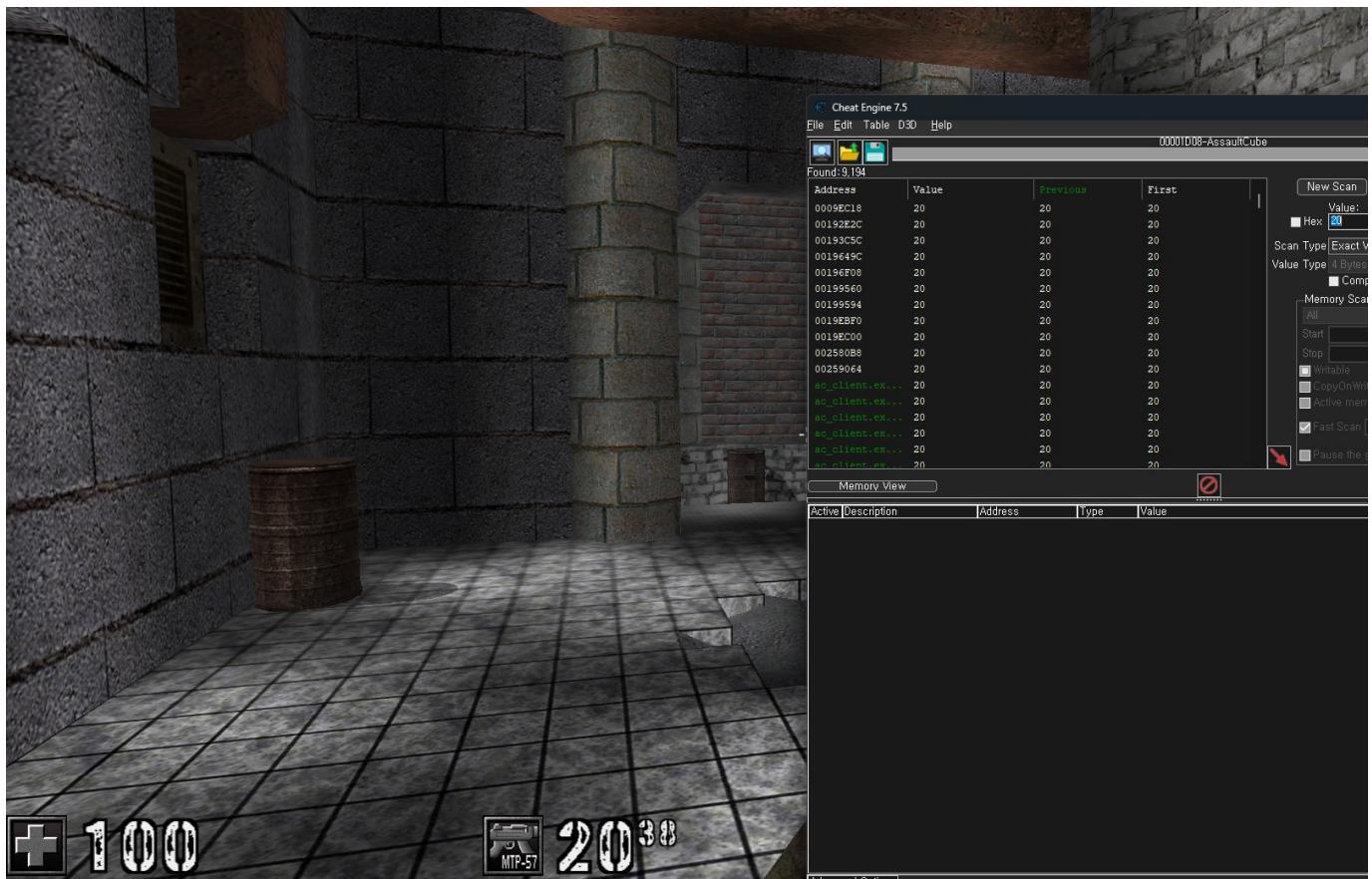
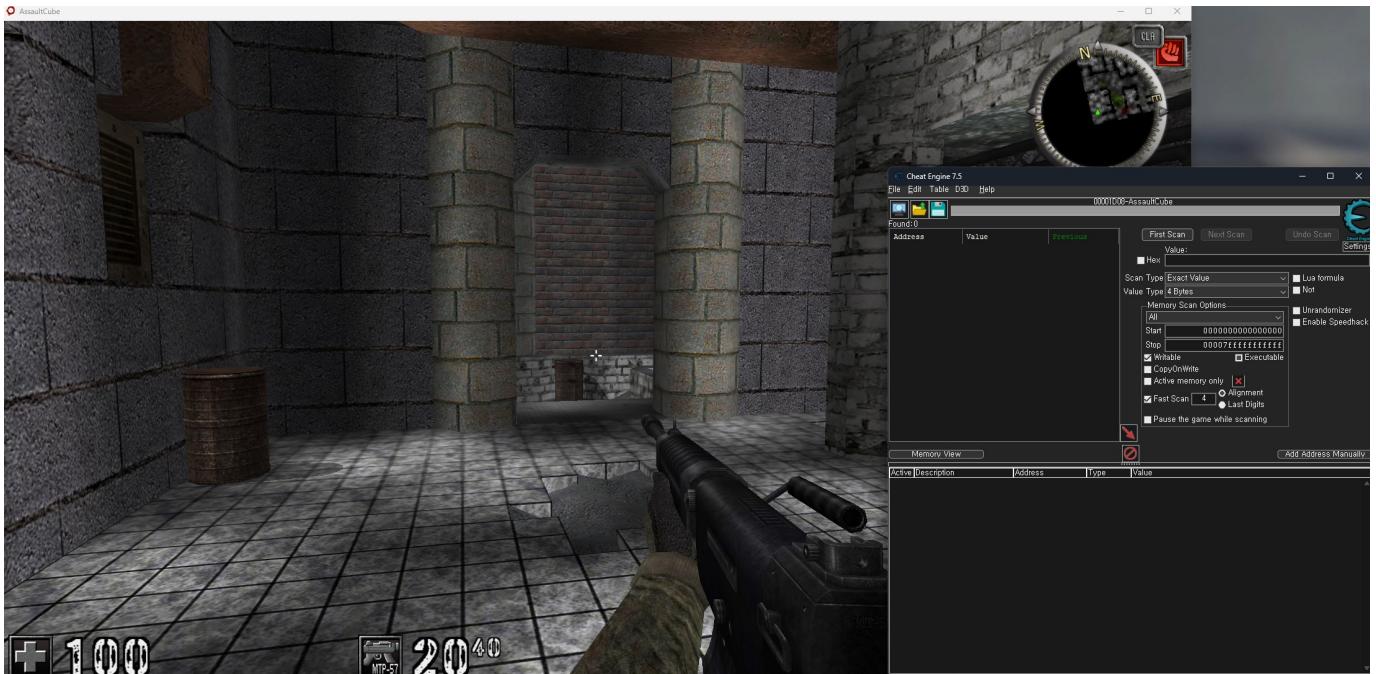
- 디버거를 잘 다룰수록 좋습니다. 수학을 잘하기 보단 수학적 사고 방식이 있다면 분석에 용이합니다. 내부구조를 분석하기 위해 해당 지식이 필요합니다.

4. Win32 API

- MS에서 개발자에게 프로그램을 만들때 쉽게 가져다 쓰라고 만들어 둔 함수들입니다. MSDN을 통해 함수의 설명서를 보고 가져와 쓰기만 하면 됩니다.

3. 메모리 해킹주소 분석

분석을 위한 툴로는 치트엔진을 사용하도록 하겠습니다.
강력한 기능들을 제공하며 사용하기 쉽습니다.



우선 총알값을 찾아 보도록 하겠습니다. 20을 스캔해 나온 무수히 많은 값에서 진짜 값을 찾기위해 총을 발사해 바뀐 총알값을 입력해 넣어 수를 줄이겠습니다.



두개의 값이 나왔으며, 둘 중 하나만 진짜입니다. 변조해서 무엇이 진짜인지 알아보면 됩니다.

첫 번째 값인 009708F0이 진짜 였군요

그렇다면 0CF6CFB4는 가짜값이니 신경쓰지 않아도 됩니다.

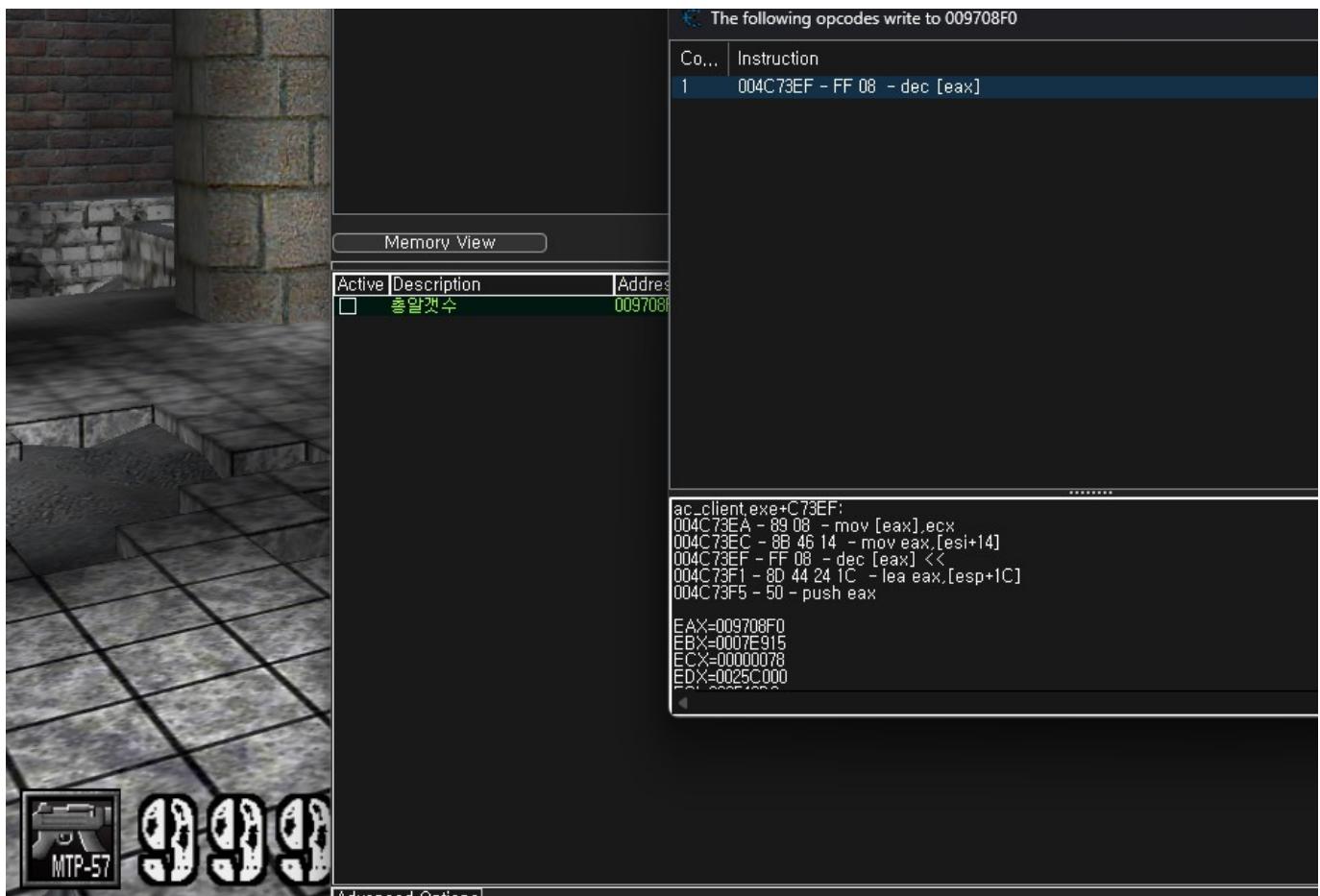


값을 찾았으나 이 값은 쓰지 못합니다. 위에서 찾은 값은 프로그램이 재시작 될 때마다 바뀌는 주소 이므로, 위 주소를 이용해 핵을 만든다 해도 타겟프로세스가 재시작되면 아무 의미가 없어집니다.

그래서 진짜값인 정적주소를 찾을 필요가 있습니다.

찾아 둔 총알값에 어떤 녀석들이 접근하는지 알아보겠습니다.
총을 발사해 총알갯수가 줄어들 때 사진과같이 카운트가 되며
FF 08 - dec [eax] 가 실행되는걸 볼 수 있습니다. 좀 더 자세히 보면,
mov eax, [esi+14] esi+14를 eax에 넣어주고 dec [eax] 즉 eax의 값을
1씩 감소 시키는 명령어입니다. 그렇다면 eax는 총알갯수라고 유추할
수 있습니다.

여러가지 방법이 있지만 현재는 두가지만 살펴보겠습니다.
첫번째로는 포인터를 찾는 것 둘째로는 직접 변조하는 것입니다.
dec [eax] 대신 1씩 증감하는 명령어인 inc [eax]로 바꾼다면 총을발사
할때 탄약이 줄어드는게 아닌 늘어날 겁니다. 또는 FF 08 을 90 90 으
로 nop nop 처리 해줌으로써 아무런 기능도 하지않게 만들어 총알이
줄어들지도 늘어나지도 않을 것입니다.



다음은 포인터스캔을 이용하는 방법입니다. 포인터를 찾는 이유는 위에서 말씀 드렸다시피 프로그램이 재시작되면 찾아둔 값의 주소는 계속 변경이 됩니다. 그렇기 때문에 포인터를 찾게 되면 타겟프로세스 안에 있는 변수의 메모리 주소를 오프셋과 함께 가리키고 있게 됩니다. 꺄다 재시작을 하게 되어도 포인터+오프셋의 구조로 바뀐 메모리의 주소의 위치를 바로 알 수 있게 됩니다.

보통 게임을 제작할때 아래와 같은 구조체를 만들고 사용하게 됩니다. 플레이어가 여러명일때 플레이어1 플레이어N 각각 선언하는건 낭비이기 때문이죠.

```
struct StatInfo
{
    int hp;
    int attack;
    int defence;
};

PlayerType playerType;
StatInfo playerStat;

MonsterType monsterType;
StatInfo monsterStat;

struct player
{
    int hp;
    int ammo;
    int delay;
    float x;
    float y;
    float z;
};
```

상단 1번째 사진은 제가 이전 TEXT RPG게임을 만들때 구현해둔 부분입니다.

우리가 현재 찾는게임의 구조체 안에는 많은 부분이 선언되어 있겠지만 저희가 찾아볼 값들만 추려서 생각해봤을때는 2번째 사진과 비슷하게 구조체가 만들어져 있을겁니다.

이와같이 보통게임은 위 코드처럼 구조체가 있고 Player는 기본적인 정보를 선언해서 나타내는 구조를 가지고 있습니다.

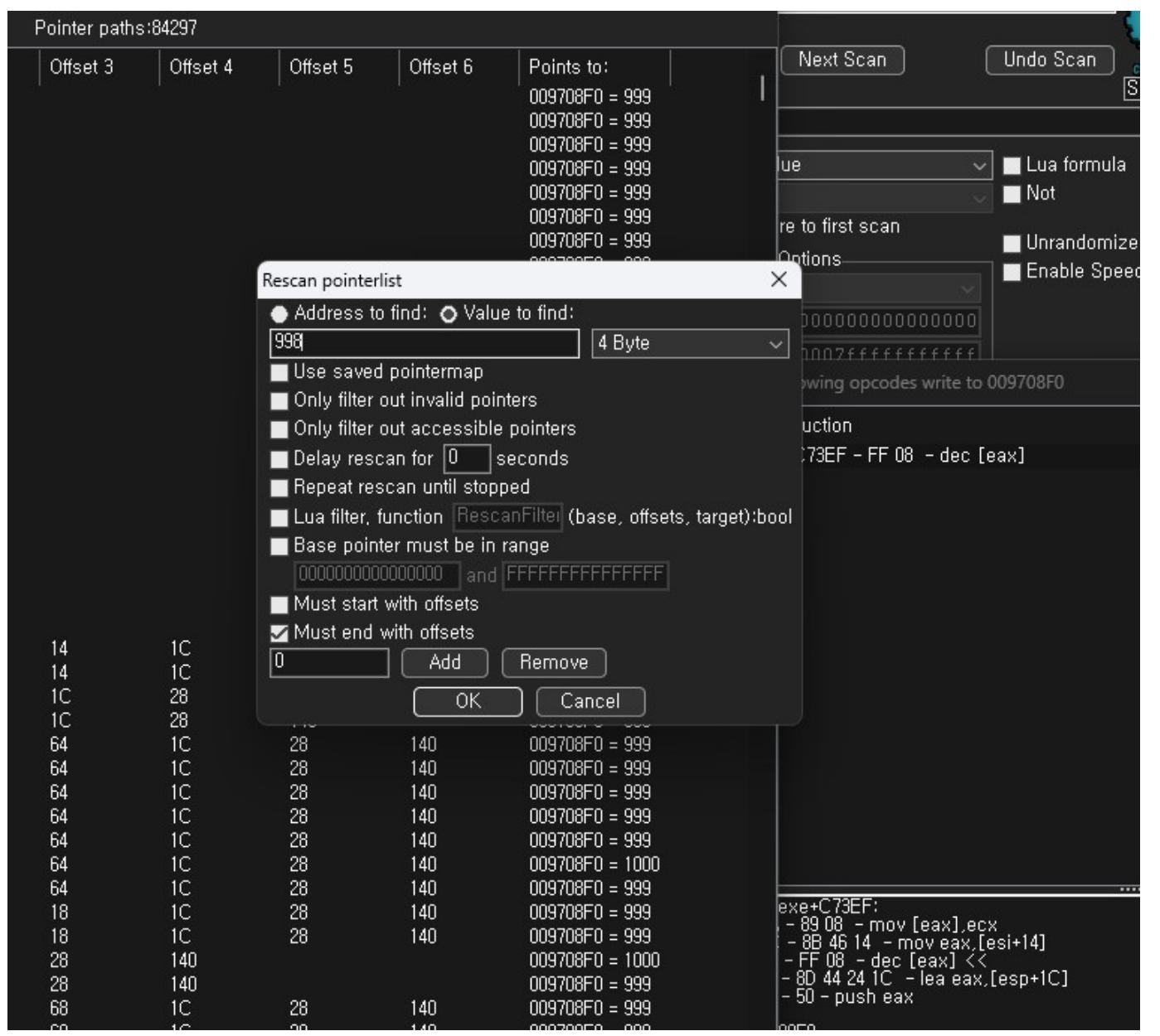
만약 위의 구조체 player 가 위게임의 프로세서인 ac_client.exe + 0x7777777 에 위치하고 있다면 hp의 정보는 0x0 만큼 떨어져 있을것이며 ammo는 0x4만큼 delay는 0x8 만큼 포인터로부터 떨어져 있을 겁니다. 그러므로 위에 찾은 값을 바탕으로 플레이어 구조체를 가리키는 포인터를 찾게 된다면 안에 들어 있는 변수들은 0x0 부터 시작해 일정 위치에 떨어져 있을 것입니다.

치트엔진은 다양한 강력한 기능들이 많이 들어있습니다.

포인터 스캔 기능을 이용해보겠습니다.

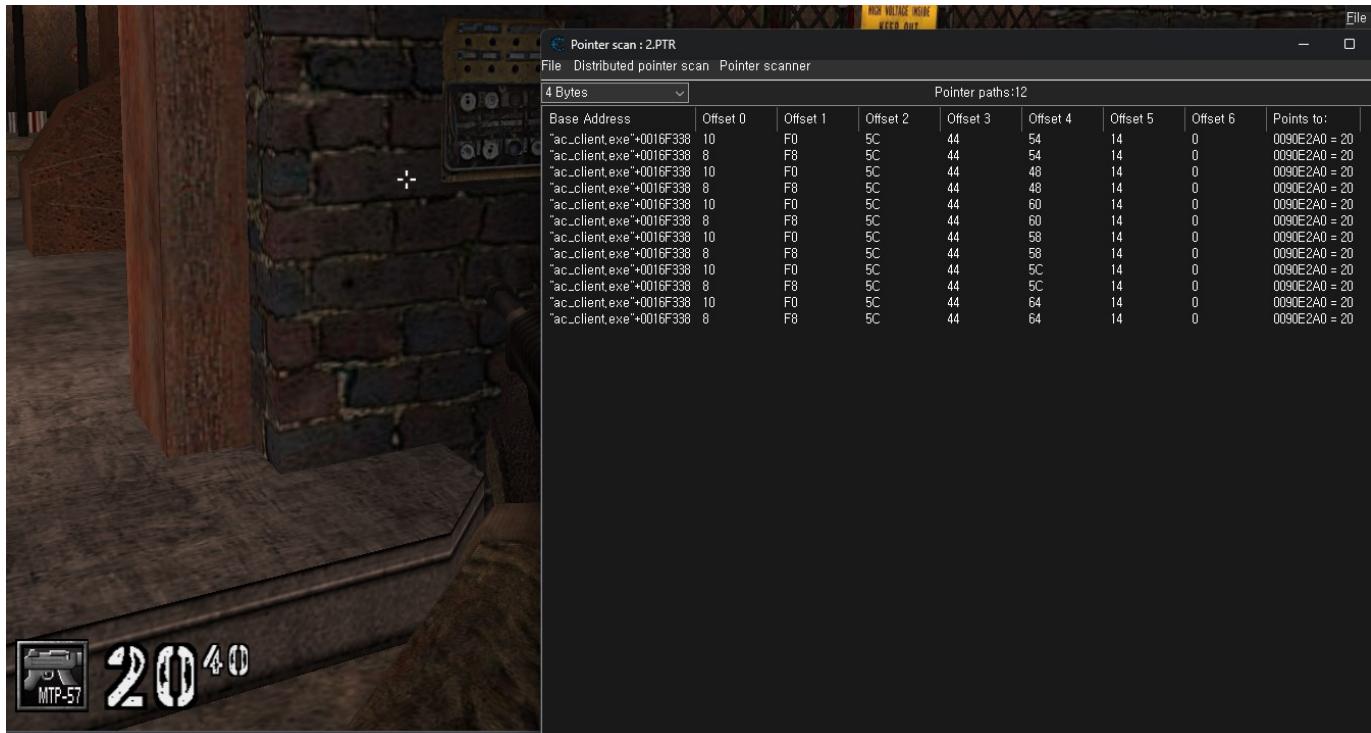
Base Address	Offset 0	Offset 1	Offset 2	Offset 3	Offset 4	Offset 5
"THREADSTACK0"-000...	138					
"THREADSTACK0"-000...	13C					
"THREADSTACK0"-000...	13C					
"THREADSTACK0"-000...	13C					
"THREADSTACK0"-000...	140					
"THREADSTACK0"-000...	140					
"THREADSTACK0"-000...	140					
"THREADSTACK0"-000...	140					

84297개의 값이 나옵니다. 제대로 동작하는 주소를 찾아야합니다. 현재 총알갯수인 998을 뱀류에 넣고 재검색을 해봅시다.

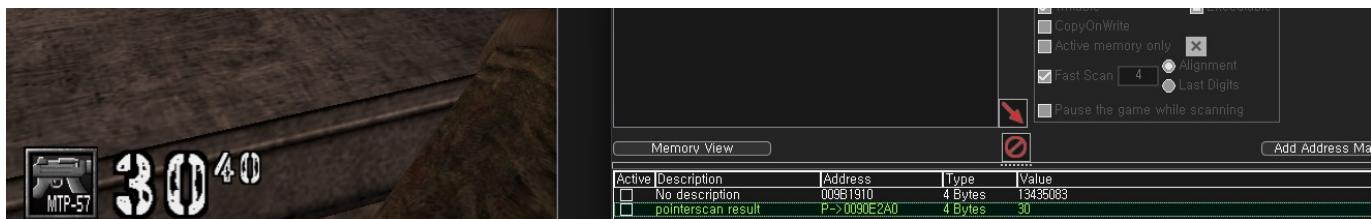


Pointer paths:115					
Base Address	Offset 0	Offset 1	Offset 2	Offset 3	Offset 4
"ac_client.exe"+000D1...	244	10	2E8	98	204
"ac_client.exe"+000D1...	228	2C	2E8	98	204
"ac_client.exe"+00043398	68	74	0		
"ac_client.exe"+0017D848	90	74	0		

115개로 줄었습니다만, 여전히 많습니다. 조금 더 줄여 보기위해 프로그램을 껐다 켜보겠습니다. 재시작을 했을 때도 주소가 제대로 살아있다면 그 값이 포인터값일 겁니다. 값이 엄청 많다면 계속해 재시작을 하는걸 추천합니다.



12개의 주소가 살아있으며 모두 총알갯수 20개로 정확히 동작하고 있는걸 볼 수 있습니다. 주소를 하나씩 테스트 해보면 됩니다만, 저 주소들은 전부 정확히 동작할 겁니다 아마 오프셋이 다를뿐 포인터는 같기 때문입니다. 5중 오프셋인걸 볼수 있으며, 특정주소에서 + offset5 해서 나온값에 다시 offset4 값을 해주고 이런식으로 타고가다보면 총알갯수가 저장되어 있는 주소가 나올겁니다.



아무 주소나 내려 값을 30으로 바꿔보니 제대로 변조되는걸 볼 수 있습니다.

0CFCC568 에서 오프셋을 타고 타고 가면 0090E2A0 이라는 값이 나옵니다.
0CFCC568 은 ac_client.exe + xxxxxx 으로 정적주소이기에 프로그램을 재시작 해도
정적주소에서 오프셋을 계속 더해주며 타고가다보면 총알갯수가 나옵니다. 0090E2A0이 아닌
다른 값이 나오겠지만 저장된 곳을 찾아가는 지도를 얻은셈입니다.

The screenshot shows the Immunity Debugger interface. On the left, there's a memory dump viewer with a red arrow pointing to a row containing the address 0090E2A0. The right side shows the search results for the address 0090E2A0. The results table has columns for Description, Address, Type, and Value. One row is highlighted in green, showing the description '탄약포인터' (ammunition pointer), address 0090E2A0, type '4 Bytes', and value 30. The status bar at the bottom displays the file path 'C:\Users\user\Desktop\ac_client.exe'.

Description	Address	Type	Value
description	009B1910	4 Bytes	13435083
탄약포인터	P->0090E2A0	4 Bytes	30
			"ac_client.exe"

체력 포인트도 찾아보겠습니다. 조금 다르게 찾는 과정이 있어 기술해보겠습니다.

The following opcodes write to 0090E24C

Co...	Instruction
1	00484499 - 89 82 EC000000 - mov [edx+000000EC],eax

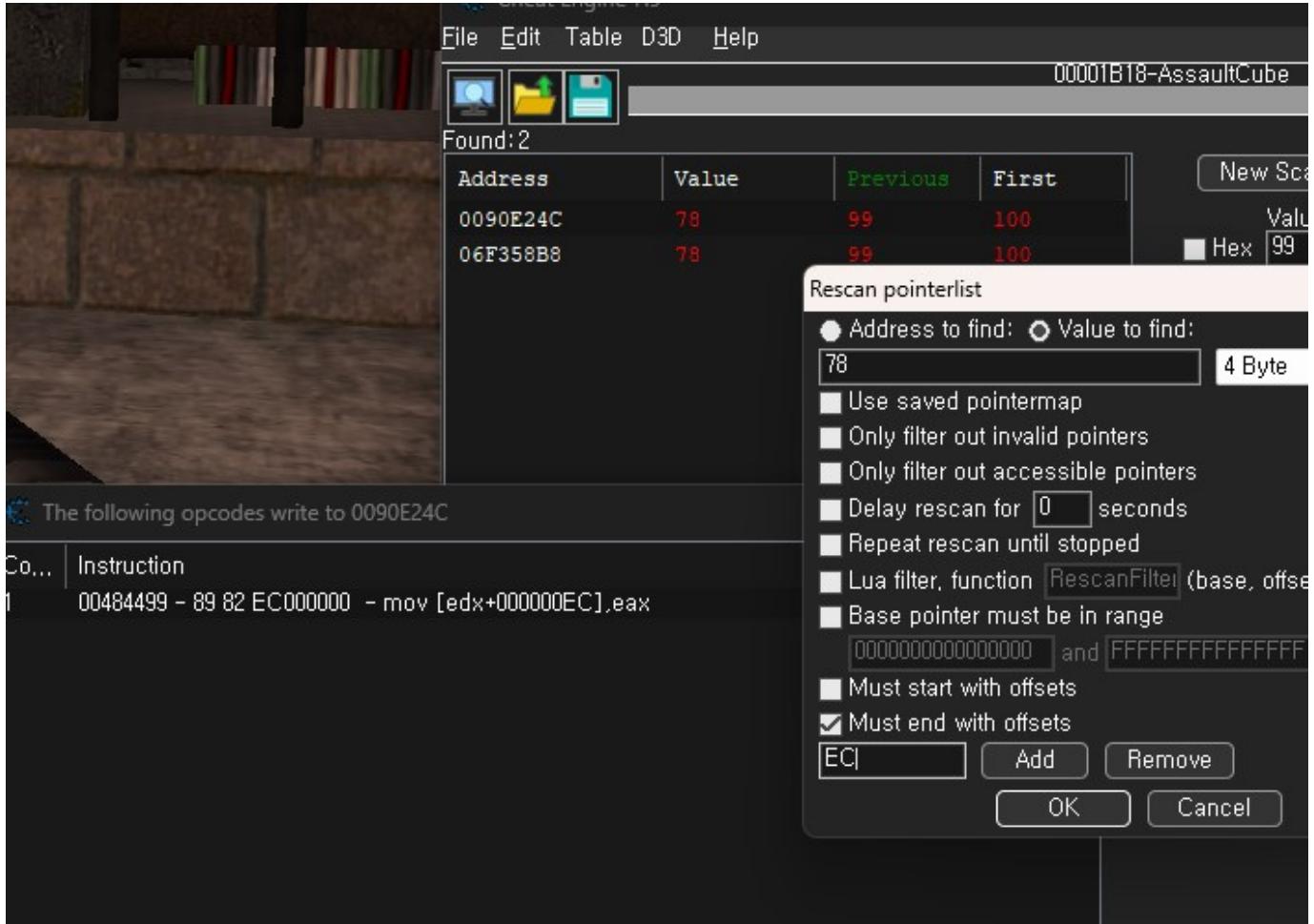
.....

ac_client.exe+84499:

```
0048449F - 89 82 F0000000 - mov [edx+000000F0],eax  
00484495 - 8B 44 24 34 - mov eax,[esp+34]  
00484499 - 89 82 EC000000 - mov [edx+000000EC],eax <<  
0048449F - 0F94 C0 - sete al  
004844A2 - 6A 00 - push 00
```

EAX=0000004E
EBX=0CFA5E40
ECX=0090E160
EDX=0090E160

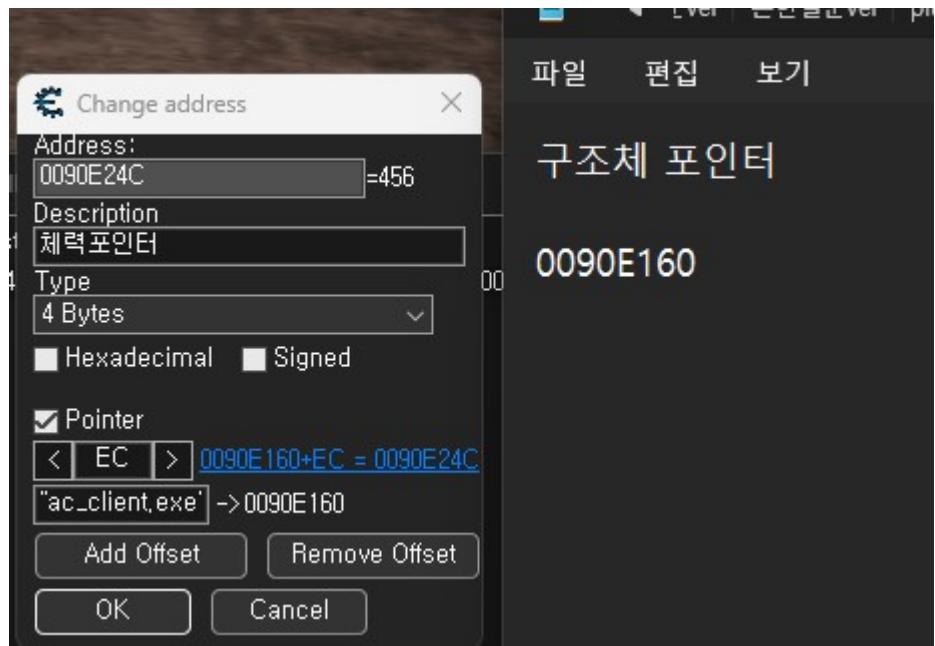
위처럼 어떤 녀석이 해당 주소에 쓰기를 하고 있는지 찾는 과정까진 같습니다만, mov [edx+0000000EC], eax 구간을 보면 eax를 edx+0000000EC의 주소에 넣어주고 있는 걸 볼 수 있습니다. eax 레지스터에는 4E라는 값이 저장되어 있고 4E를 10진수로 바꾸면 78이 됩니다. 78은 현재의 체력값이며 78을 edx+0000000EC의 주소에 넣어준다는 뜻이 됩니다. 그렇다면 우리가 짐작 할 수 있는건 포인터에서 0xEC의 오프셋만큼 떨어진 곳이 체력이 저장되는 곳이겠네요.



똑같이 포인트스캔을 해주겠습니다. Value to find에는 현재 체력값인 78을 넣습니다. 이전 총알포인터를 찾을 때는 Must end with offsets 를 0으로 설정 했습니다, 지금은 EC를 넣어주고 스캔해주면 됩니다.

그 이후의 방법은 위에서 설명한 방법과 동일합니다. 값을 줄여나가며 포인터를 찾아줍시다.

Active	Description	Address	Type	Value
<input type="checkbox"/>	탄약포인터	P->0090E2A0	4 Bytes	22
<input type="checkbox"/>	체력포인터	P->0090E24C	4 Bytes	456

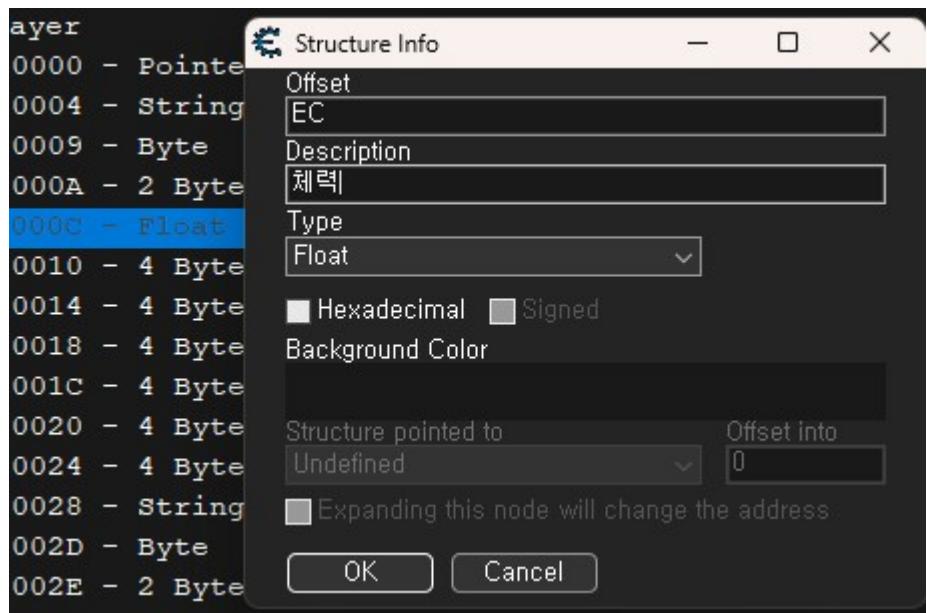


0090E160의 위치에서 EC만큼 떨어진 곳에 체력값이 저장되어 있군요
 0090E160이란 값은 재시작 할때마다 바뀔 것입니다. 가려져 있지만 그 옆의
 ac_client.exe+0017E0A8 의 정적주소는 바뀌지 않을 것입니다. 정리하자면
 ac_client.exe+0017E0A8 + 0xEC 에 체력값이 저장되며 플레이어구조체 주소는
 ac_client.exe+0017E0A8 이란 말입니다.

위에서 보았던 mov [edx+0000000EC], eax 에서 EAX레지스터에는 78이란 체력값이
 EDX 레지스터에는 0090E160이 저장되어 있었습니다. 이미 찾은 것이나 다름이 없었지만
 정적주소인 포인터는 몰랐으니 괜찮습니다.

Structure dissect:Player	
Offset-description	Address: Value
Player	
0000 - Pointer	90E160 : P->0054D0A4
0004 - String	90E164 : ;#MCG
0009 - Byte	90E169 : 216
000A - 2 Bytes	90E16A : 17080
000C - Float	90E16C : 4.5
0010 - 4 Bytes (Hex)	90E170 : 8000000C
0014 - 4 Bytes (Hex)	90E174 : 8000000C
0018 - 4 Bytes	90E178 : 12
001C - 4 Bytes	90E17C : 0
0020 - 4 Bytes	90E180 : 0

구조체를 보겠습니다. 0090E160 을 넣어주면 그 밑으로 무언가 많이 나오는걸 볼 수 있습니다.
 이곳에 플레이어의 좌표, 이름, 총, 탄약등 많은 변수들이 저장되어 있겠네요



EC번째는 체력이 저장되어 있을겁니다. 구분하기 쉽게 EC번째의 이름을 체력으로 바꿔주고 이동해보면 현재의 체력값인 456이 4바이트 타입으로 저장되어 있는걸 볼 수 있습니다.

Offset-description	Address: Value
00CC - 4 Bytes	90E22C : 4294966944
00D0 - Float	90E230 : 100
00D4 - 4 Bytes	90E234 : 955
00D8 - 4 Bytes	90E238 : 955
00DC - Pointer	90E23C : P->06FA4910
00E0 - Pointer	90E240 : P->0C83EC98
00E4 - 4 Bytes	90E244 : 1294
00E8 - Pointer	90E248 : P->0054D0B4
00EC - 4 Bytes	90E24C : 456
00EC - 체력	90E24C : 456
00F0 - 4 Bytes	90E250 : 0
00F4 - 4 Bytes	90E254 : 6
00F8 - 4 Bytes	90E258 : 6
00FC - 4 Bytes	90E25C : 6
0100 - 4 Bytes	90E260 : 0
0104 - 4 Bytes	90E264 : 1
0108 - 4 Bytes	90E268 : 50
010C - 4 Bytes	90E26C : 0
0110 - 4 Bytes	90E270 : 0

ac_client.exe+2E638	89 77 04	mov [edi+04],esi
ac_client.exe+2E63B	89 4F 08	mov [edi+08],ecx
ac_client.exe+2E63E	33 C9	xor ecx,ecx
<hr/>		
Protect:Read/Write AllocationBase=008E0000 Base=0090E000 Size=94000		
address 60 61 62 63 64 65 66 67	68 69 6A 6B 6C 6D 6E 6F	70 71 72 73
0090E160 A4 D0 54 00 3B 23 4D 43	47 D8 B8 42 00 00 90 40	0C 00 00 80
0090E188 3B 23 4D 43 47 D8 B8 42	00 00 00 00 C3 E8 24 43	C0 8C FE C0
0090E1B0 00 00 90 40 00 00 90 40	33 33 33 3F 00 01 00 00	00 00 00 00
0090E1D8 00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	00 00 00 00
0090E200 00 00 00 00 00 00 00 00	00 00 C8 42 00 01 02 00	00 00 00 00
0090E228 28 00 00 00 A0 FE FF FF	00 00 C8 42 BB 03 00 00	BB 03 00 00

메모리뷰로 이동해 0090E160 으로 이동해줍니다. HEX로 보여주고 있어 보기 힘이듭니다.
Display Type 을 10진수로 바꿔줍니다.

call procedure									
Protect:Read/Write	AllocationBase=008E0000	Base=0090E000	Size=94000						
address 60	64	68	6C	70	74	78	7C	80	84
0090E160 5558436	1129128763	1119410247	1083179008	2147483660	2147483660	12	0	0	0
0090E188 1129128763	1119410247	0	1126492355	3237907648	2147483648	2147483652	1098907648	0	1066192077
0090E1B0 1083179008	1083179008	1060320051	256	0	257	0	1128792064	0	0
0090E1D8 0	0	0	0	0	0	0	1120403456	0	0
0090E200 0	0	1120403456	131328	0	40	4294966944	1120403456	131328	0
0090E228 40	4294966944	1120403456	955	955	117066000	209972376	1294	5558452	456
0090E250 0	6	6	6	0	1	50	0	0	0
0090E278 0	40	0	0	1	10	0	0	0	0
0090E2A0 22	0	0	0	0	0	0	0	0	0
0090E2C8 0	0	0	0	0	0	0	0	8	6
0090E2F0 0	0	0	0	0	0	0	0	22	0
0090E318 0	2	0	0	0	0	0	0	0	0
0090E340 0	0	0	515815	517762	517180	0	793	0	1634628864
0090E368 1684368754	0	0	0	0	0	0	0	0	0
0090E390 0	0	0	0	0	0	0	0	0	0
0090E3B8 0	0	0	0	0	0	0	0	0	0

드래그 친곳에 456 체력값이 저장되어 있네요 이곳을 보게되면 장전된 총알갯수, 남은총알갯수
값도 보이네요 float 형식으로 보게되면 x, y, z 좌표도 보이겠네요

우선 위에 찾았던 탄약포인터는 5중이였고 잘 쓰이지 않습니다. c언어에서도 한다면 N중 포인
터까지 쓰겠지만 보통 2중포인터까지만 활용하는 것 처럼 말이죠

현재 총알값은 0090E2A0에 저장되어 있네요 오프셋을 알기 위해선 총알이 저장된주소 - 구조체
주소를 해주면 나오겠네요 즉, 0090E2A0 - 0090E160 -> 140 이 됩니다.

총알갯수는 플레이어 구조체포인터에서 0x140 번째에 떨어져 있습니다. 이동해보면
제대로 들어가 있는 모습을 볼 수 있습니다.

-0120 - 4 Bytes	90E280 : 0
-0124 - 4 Bytes	90E284 : 0
-0128 - 4 Bytes	90E288 : 1
-012C - 4 Bytes	90E28C : 10
-0130 - 4 Bytes	90E290 : 0
-0134 - 4 Bytes	90E294 : 0
-0138 - 4 Bytes	90E298 : 0
-013C - 4 Bytes	90E29C : 0
-0140 - 4 Bytes	90E2A0 : 18
-0140 - 탄약	90E2A0 : 18
-0144 - 4 Bytes	90E2A4 : 0



좌표를 찾는 법은 쉽다. float 타입으로 디스플레이 타입을 바꾸고 캐릭터를 움직여보면 계속해서 변하는 값이 있다. 플레이어의 y좌표로 보이는 곳[드래그 한 곳]을 30.00으로 바꿔보니 맵을 뚫고 하늘로 치솟았다. 이런식으로 x,y,z 좌표를 구할 수 있고 바라보는 방향을 마우스로 돌리며 캐릭터가 어딜 보고 있는지도 구할 수 있다. 좌표의 경우 월핵, 에임봇에 쓰이는게 일반적이다.

Offset-description	Address: Value
0009 - Byte	90E169 : 0
000A - 2 Bytes	90E16A : 16956
0014 - 4 Bytes (Hex)	90E174 : 8000000C
0018 - 4 Bytes	90E178 : 12
0020 - 4 Bytes	90E180 : 0
0024 - 4 Bytes	90E184 : 0
0028 - 캐릭터 X좌표	90E188 : 182.8886414
0028 - String	90E188 : ~?6C
002C - 캐릭터 Z좌표	90E18C : 47
002D - Byte	90E18D : 0
002E - 2 Bytes	90E18E : 16956
0030 - 캐릭터 Y좌표	90E190 : -1
0030 - 4 Bytes	90E190 : 3212836864

4. 해킹툴 개발

리버스 엔지니어링을 통해 동작중인 게임의 메모리주소를 얻은 후 프로세스에 접근 하는 방식은 내부접근과 외부접근으로 두가지의 방식이 있습니다.

Internal (내부접근)

- 프로세스에 DLL(동적라이브러리)를 주입(Injection) 하는 DLL - Injection 기법을 사용합니다.
직접적으로 주입하는 것이기 때문에, 메모리에 직접 접근이 가능해 빠르게 읽고 쓸 수 있으며, 코드 또 한 보다 간결하며 강력합니다. 아래 GetModuleHandle 함수를 사용했으나 정적코딩도 가능합니다.

```
DWORD WINAPI siyul(void)
{
    DWORD memoryA = (DWORD)GetModuleHandleA("target module") + 0x123456;
    DWORD memoryB = *(DWORD*)(memoryA)+0x123;
    int changeValue = 999;
    CopyMemory((LPBYTE)memoryB, &changeValue, 4);
}
```

External (외부접근)

- Win32 API를 이용해 프로세스의 메모리와 작용합니다. DLL(동적라이브러리)을 직접적으로 주입(Injection) 하는것이 아닌 EXE 실행파일을 주로 사용합니다. Kernel32!OpenProcess() 를 이용해 프로세스의 접근권한(HANDLE)을 받아오고 ReadProcessMemory(), WriteProcessMemory() 함수를 통해 메모리와 상호작용 합니다. 이와 같은 방식은 Internal 방식 보다는 커널(ring0)에 API 호출도 해야하므로 처리 시간이 걸려 속도가 느립니다. 예를들어 OpenProcess() 호출시
Usermode(ring3)에서 Kernel32!OpenProcess -> ntdll!ZwOpenProcess > ntdll!KiFastSystemcall
-> SYSENTER(명령) -> msr(레지스터) 를 거쳐
Kernelmode(ring0)에 접근합니다. 그 후 이어서
-> nt!KiFastCallEntry -> nt!NtOpenProcess 를 가져옵니다.

네 아무튼 느릅니다.

```
int main()
{
    HWND hWnd = FindWindow(NULL, L"TitleName");
    DWORD targetProc = GetWindowThreadProcessId(hWnd, &procID);
    HANDLE hHandle = OpenProcess(PROCESS_ALL_ACCESS, 0, procID);
    VirtualProtectEx(hHandle, (LPVOID*)(GetModuleBaseAddress(procID, L"target module") + 0x123456), 2, PAGE_EXECUTE_READWRITE, &oldProtect);
    ReadProcessMemory(hHandle, (LPVOID*)(GetModuleBaseAddress(procID, L"target module") + 0x123456), &addr, sizeof(addr), NULL);
    addr += 0x123;
    changeValue = 999;
    WriteProcessMemory(hHandle, (LPVOID*)addr, &changeValue, sizeof(changeValue), NULL);
    VirtualProtectEx(hHandle, (LPVOID*)(GetModuleBaseAddress(procID, L"target module") + 0x123456), 2, oldProtect, &oldProtect);
}
```

고작 4줄에 불과하며, 정적으로 주소값 입력시 API이용조차 하지 않아도 되는 Internal 방식에 비해 External 방식은 핸들을 구하고 PID를 구해 접근권한을 얻어 메모리보호 속성을 변경해주고 메모리를 읽어와 오프셋을 더한 후 메모리를 변조해줍니다. 그리곤 보호속성을 원래대로 돌려줍니다.

본 문서에선 DLL Injection 기법을 사용하는 Internal 방식이 아닌 External 방식을 이용하겠습니다.

```

int main()
{
    SetConsoleTitle(TEXT("MyHackTool"));
    cout << "[!] 타겟 프로세스를 기다리는 중... :F" << endl;
    while (true)
    {
        Sleep(100);
        HWND hWnd = FindWindow(NULL, L"AssaultCube");
        if (hWnd)
        {
            cout << "[1] 프로세스를 찾았습니다 !! :D" << endl;
            DWORD targetProcess = GetWindowThreadProcessId(hWnd, &procID);
            if (targetProcess)
            {
                cout << "[2] PID를 가져오는데 성공했습니다 !! :D" << endl;
                HANDLE hHandle = OpenProcess(PROCESS_ALL_ACCESS, 0, procID);
                |
            }
        }
    }
}

```

Sleep() 을 통해 과부하를 막아주었습니다. 1000 = 1초입니다.

FindWindow() 는 클래스 이름 또는 캡션 이름으로 원하는 윈도우창의 핸들 값을 얻는 함수입니다. 찾은 경우 핸들값을 반환하고 찾지 못한다면 NULL값을 반환합니다.

GetWindowThreadProcessId() 는 HWND 값을 이용해 PID를 알려주는 함수입니다. 첫번째 파라미터로는 PID를 얻고자하는 윈도우의 핸들을 넣어주면 되며 두번째 인자로는 반환받을 PID의 포인터입니다.

OpenProcess()는 프로세스의 핸들값을 얻어올때 사용합니다. 성공시 핸들값을 반환합니다. 첫번째 파라미터로는 접근유형이며 제일 많이 사용하는건 PROCESS_ALL_ACCESS로 모든 권한을 가질 수 있습니다. 두번째 파라미터는 현재 함수를 프로세스에 상속할지 아닐지 결정하는 인자이지만 보통 0을 사용합니다. 세번째 파라미터는 PID입니다. 0또는 NULL을 넣을시 모든 프로세스에 접근합니다.

이제 다음은 변조할 모듈의 베이스주소를 구해야 합니다. 정적으로 입력해줄수 있습니다만 함수를 이용해 구하겠습니다. 베이스주소를 구하는 이유는 예를들어 A프로그램이 실행되면 A프로그램.exe + 400000 ~ 799999 까지는 A프로그램.exe 모듈의 주소라 치겠습니다. 프로그램을 실행하는데 필요한 b.dll도 로드가 되어있다 한다면 b.dll + 800000 ~ 부터 시작 하는 것 처럼 각자의 공간에 모듈들이 메모리에 올라가 있습니다.

변조할 포인터들은 게임클라이언트에 올라가 있었으니 클라이언트의 베이스주소를 구합니다.

```

DWORD GetModuleBaseAddress(DWORD pId, const wchar_t* moduleName)
{
    uintptr_t modBaseAddr = 0;
    HANDLE hSnap = CreateToolhelp32Snapshot(TH32CS_SNAPMODULE | TH32CS_SNAPMODULE32, pId);
    if (hSnap != INVALID_HANDLE_VALUE)
    {
        MODULEENTRY32 modEntry;
        modEntry.dwSize = sizeof(modEntry);
        if (Module32First(hSnap, &modEntry))
        {
            do
            {
                if (!_wcsicmp(modEntry.szModule, moduleName))
                {
                    modBaseAddr = (uintptr_t)modEntry.modBaseAddr;
                    break;
                }
            } while (Module32Next(hSnap, &modEntry));
        }
        CloseHandle(hSnap);
        return modBaseAddr;
    }
}

```

GetModuleBaseAddress()를 만들어 주었습니다. 첫번째 인자로는 pid를 두번째 인자로는 구할 모듈의 이름을 넣습니다.

```

C++                                복사

HANDLE CreateToolhelp32Snapshot(
    [in] DWORD dwFlags,
    [in] DWORD th32ProcessID
);

```

CreateToolhelp32Snapshot()에 대한 정보를 msdn에서 가져왔습니다.

-_In_DWORD dwFlags

스냅샷에 포함할 정보, 이 매개 변수는 다음 값 중 하나 이상일 수 있습니다.

_In_DWORD th32ProcessID

스냅샷에 포함할 프로세스의 프로세스 식별자 (프로세스 아이디 넣으면 됩니다.)

TH32CS_SNAPMODULE | TH32CS_SNAPMODULE32 를 넣어 모듈 스냅샷을 구해옵니다.

구조체를 사용하기위해 MODULEENTRY32 dwSize 를 초기화 해줍니다.

Module32First()로 모듈에 대한 정보를 검색합니다. szModule로 내가 찾는 모듈이름과 맞는지 비교합니다. 맞다면 modBaseAddr로 베이스주소를 저장합니다.

Module32Next()로 맞는프로세스를 찾을때까지 다음으로 넘어갑니다.

MODULEENTRY32 구조체의 사진은 아래에 기술합니다.

```
C++ 복사

typedef struct tagMODULEENTRY32 {
    DWORD dwSize;
    DWORD th32ModuleID;
    DWORD th32ProcessID;
    DWORD GblcntUsage;
    DWORD ProcntUsage;
    BYTE *modBaseAddr;
    DWORD modBaseSize;
    HMODULE hModule;
    char szModule[MAX_MODULE_NAME32 + 1];
    char szExePath[MAX_PATH];
} MODULEENTRY32;
```

MSDN에 자세히 나와 있습니다.

```
        cout << "[2] PID를 가져오는데 성공했습니다 !! :D" << endl;
        HANDLE hHandle = OpenProcess(PROCESS_ALL_ACCESS, 0, procID);
        BASE_ADDR = GetModuleBaseAddress(procID, L"ac_client.exe");
        PLAYER_BASE = BASE_ADDR + 0x0017E0A8;
        cout << "Target PID : " << procID << endl << "ModuleBaseAddr : " << BASE_ADDR;
        HotKeyActivate = true;
        break;
    }
}

if (HotKeyActivate == true)
{
    cout << "===== HOT KEYS =====" << endl;
    cout << "[NUMPAD 1] : HP 999" << " || " << "[NUMPAD 2] : 탄약무한" << endl;
    cout << "[NUMPAD 3] : 노딜레이" << " || " << "[NUMPAD 4] : 총기반동제거" << endl;
    while (true)
    {
        Sleep(100);
        if (GetAsyncKeyState(VK_NUMPAD1))
        {
        }
    }
}
```

ac_client.exe의 베이스 주소를 가져와 BASE_ADDR에 저장해주었습니다. 그 후 PLAYER_BASE에 BASE_ADDR + 0x0017E0A8을 해주었습니다. 위에서 구했던 플레이어 구조체의 정적주소입니다.

GetAsyncKeyState() 함수를 이용해 해당키의 입력이 들어오면 맞는 기능을 실행하게 해주면 되겠군요

```

Sleep(100);
if (GetAsyncKeyState(VK_NUMPAD1))
{
    DWORD oldProtect;
    VirtualProtectEx(hHandle, (BYTE*)PLAYER_BASE, 1, PAGE_EXECUTE_READWRITE, &oldProtect);
    ReadProcessMemory(hHandle, (BYTE*)PLAYER_BASE, &pAddress, sizeof(pAddress), NULL);
    pAddress += 0xEC;
    newValue = 999;
    WriteProcessMemory(hHandle, (BYTE*)pAddress, &newValue, sizeof(newValue), NULL);
    VirtualProtectEx(hHandle, (BYTE*)PLAYER_BASE, 1, oldProtect, &oldProtect);
}

if (GetAsyncKeyState(VK_NUMPAD2))
{
    DWORD oldProtect;
    VirtualProtectEx(hHandle, (BYTE*)(BASE_ADDR + 0xC73EF), 2, PAGE_EXECUTE_READWRITE, &oldProtect);
    WriteProcessMemory(hHandle, (BYTE*)(BASE_ADDR + 0xC73EF), (BYTE*)"\\x90\\x90", 2, NULL);
    VirtualProtectEx(hHandle, (BYTE*)(BASE_ADDR + 0xC73EF), 2, oldProtect, &oldProtect);
}

```

기능을 넣어주었습니다.

VirtualProtectEx() 함수를 이용 -> 변조할 메모리영역의 보호 상태를 변경해줍니다.

ReadProcessMemory() 함수를 이용 -> PLAYER_BASE (플레이어구조체포인터)를 읽어와 주소를 저장해줍니다. + 0xEC (위에서 찾은 체력위치 오프셋) 를 해줍니다.
바꿀 체력값은 999로 설정해준 뒤

WriteProcessMemory() -> 플레이어 구조체포인터 + 0xEC를 해준 위치에 새로운 체력값인 999를 덮어 씁니다.

VirtualProtectEx() 를 다시 사용해 메모리 보호속성을 원래대로 돌려줍니다.

체력 변조 부분은 미리 구해놨던 플레이어 구조체 포인터에 오프셋을 더해 구현했습니다. 데미지를 받는 부분을 Detour Hooking 을 이용해 Trampoline Patch를 해서 할 수 있을거 같습니다만 이번 문서에서는 다루지 않겠습니다.

총알무한 부분은 오프셋을 사용해 총알갯수를 변경하기보단

89 08 - mov [eax], ecx	// ecx값을 eax의주소에 넣는다.
8B 46 14 - mov eax, [esi+14]	// esi+14의 주소에 저장된 값을 eax에 넣는다.
FF 08 - dec, eax	// eax값을 1씩 감소시킨다. <- 총알갯수
8D 44 24 1C - lea eax, [esp+1C]	// [esp+1C] 주소값을 eax에 저장한다.

FF 08 - dec, eax -> 90 90 - nop nop

으로 변경해주며 총알갯수가 줄어들지도 늘어나지도 않도록 하였습니다.

nop 명령어는 아무동작도 하지않는 명령어입니다. 물론 inc, eax 로 총알갯수를 1씩 늘릴수도 있습니다만 nop을 사용하여 총알이 줄어드는 기능을 아무동작도 하지않게 만들었습니다.

메모리에 접근하는 부분을 구현하고나니 보기 가 좀 그렇습니다. 따로 함수를 만들어 메모리와 작용하는 부분을 합쳐보겠습니다. HackActivate로 해볼까요

```
namespace memory
{
    void HackActivate(HANDLE hProc, BYTE* addr, BYTE* buf, unsigned int size);

void memory::HackActivate(HANDLE hProc, BYTE* addr, BYTE* buf, unsigned int size)
{
    DWORD oldProtect;
    VirtualProtectEx(hProc, addr, size, PAGE_EXECUTE_READWRITE, &oldProtect);
    WriteProcessMemory(hProc, addr, buf, size, NULL);
    VirtualProtectEx(hProc, addr, size, oldProtect, &oldProtect);

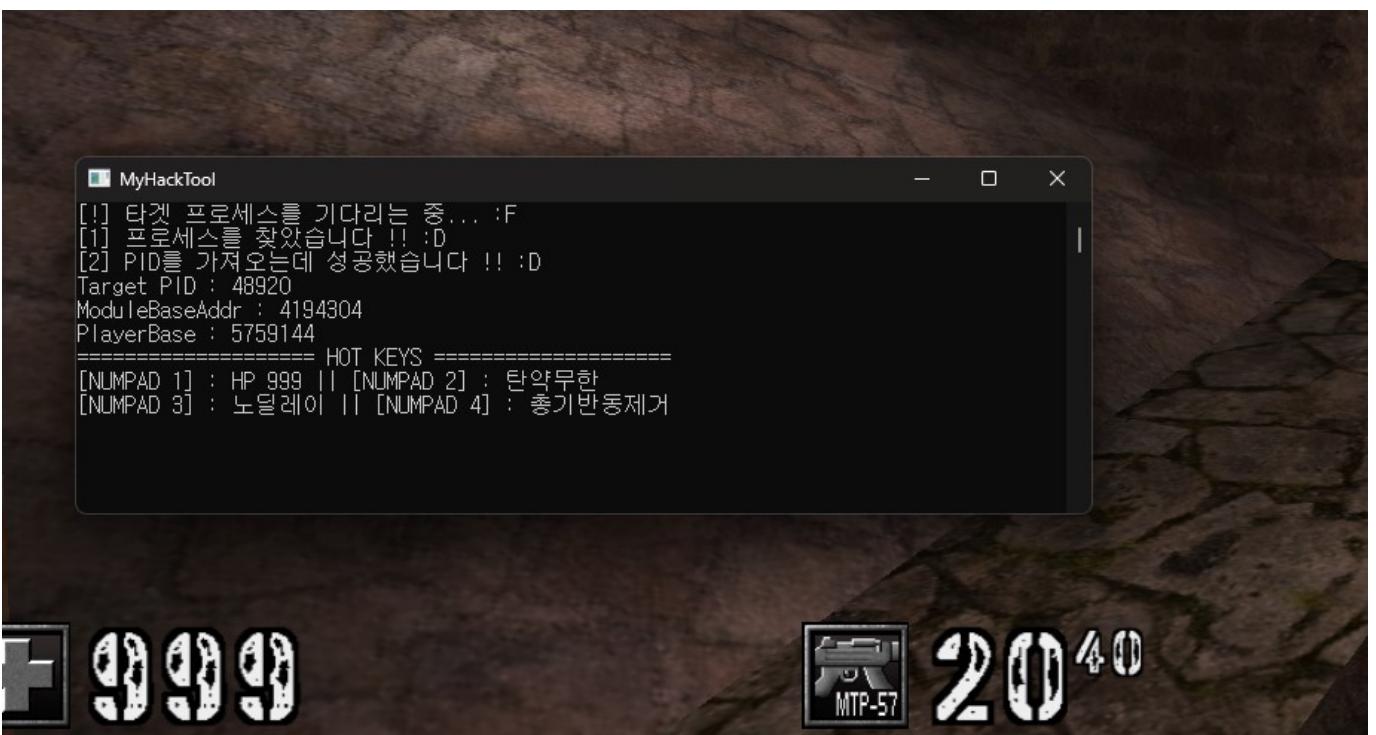
    if (GetAsyncKeyState(VK_NUMPAD2))
    {
        memory::HackActivate(hHandle, (BYTE*)(BASE_ADDR + 0xC73EF), (BYTE*)"\\x90\\x90", 2);
    }
}
```

한결 깔끔해 졌습니다. 전체적으로 소스코드를 간결하게 다듬어 줄 수 있겠지
이제 기능이 잘 동작하는지 확인해 볼 차례입니다.





정상적으로 동작하는 모습을 볼 수 있습니다.
메모리 변조 또한 제대로 동작합니다.
사진으로는 체력999 말고는 보여드릴수가 없지만 4개의 기능 모두 동작하는걸
확인했습니다.



5. 끝맺음

실력이 많이 부족하고 글 쓰는 재주까지 없지만 좋게 봐주셨으면 고맙겠습니다.

월핵의 경우 2가지의 방법을 많이 사용하며

1. Direct3D Hooking을 하는 방법 (Internal 방식)

- 가상함수들의 주소를 담는 vTable의 시작주소를 알아낸 후, 오프셋 계산을 통해 EndScene 함수의 주소를 찾아낸 후 패치를 진행합니다. EndScene 함수는 장면의 렌더링을 종료할 때 호출 되므로 myHook 함수를 구현한 뒤 EndScene 함수가 호출 될때 myHook 함수로 점프시켜 기능을 실행한 뒤 EndScene 함수를 호출 해주면 됩니다.

위의 과정은 Detours Hooking 또는 Inline Function Hooking이라 불리며 매우 강력한 기능을 가진 후킹 방법입니다.

Direct3D 에 관한 지식과 후킹에 관한 지식도 알아야하므로 선행해야 할 지식이 매우 많습니다.

2. 오버레이 이용 (External 방식)

GDI(Graphics Device Interface) : 윈도우에서 그래픽 작업 처리를 위해 제공하는 API 를 사용합니다.

컴파일하면 보여지는 검은바탕의 콘솔창을 함수를 GDI API를 이용하여 창의 배경을 투명하게, 어떤 상황에서든 창이 제일 위에 위치하게, 최소화 최대화 종료 버튼삭제 등을 한 뒤 타겟프로세스의 창크기를 읽어와 창 위에 딱맞게 부착시킨다.

그리고 플레이어와 에너미 xyz 좌표를 읽어와 계산해 그려준다.

메모리를 읽어오기만 하면되므로 변조할 필요가 없다.

변조할 필요가 없다는 건 보안솔루션을 우회하지 않아도 된다는 말이됩니다.

그리고 본 문서에서 타겟프로그램으로 삼은 게임은 어떠한 보안도 적용되어 있지 않습니다.

온라인게임등에서는 코드들의 가상화와 더불어 불법프로그램을 탐지하는 함수들 그리고 서드파티 보안솔루션들이 동작하므로 쉽게 분석되지 않으며, 분석을 하기위한 툴을 사용하는 과정또한 쉽지 않습니다. 보안솔루션들은 루트킷처럼 커널권한에서 동작하는 드라이버를 사용하기도 하며, 여러 보안기법들로 무장해있습니다.

그럼에도 유저들이 게임을 즐길때 핵이 판치는 이유는, 영원한 창과 방패의 싸움이며 창이 절대적으로 유리하다는 점입니다.

여러 보안기법들은 그저 해커들이 좀 더 어렵게 시간을 끄는 용도라 보여지는게 현실입니다.

모든 값을 서버에서 관리하는 게임이라면 메모리변조를 통한 핵은 없습니다만, 보통의 게임들 특히 FPS 게임같이 1ms를 중요시하는 게임이라면 HOST에게 어느정도의 권한을 부여하고 클라이언트에 연산을 맡기는 편입니다. 그렇기에 클라이언트 변조를 통해 핵이 사라지지 않는 것입니다.

여담으로 어도비 pdf문서로 작성하였는데 프로그램이 상당히 많이 무거워 애먹었네요

마무리하겠습니다. 감사합니다.

main.cpp

```
#include <stdio.h>
#include <Windows.h>
#include <iostream>
#include <tlhelp32.h>
#include "offset.h"
#include "memory.h"

int main()
{
    SetConsoleTitle(TEXT("MyHackTool"));
    cout << "[!] 타겟 프로세스를 기다리는 중... :F" << endl;
    while (true)
    {
        Sleep(100);
        HWND hWnd = FindWindow(NULL, L"AssaultCube");
        if (hWnd)
        {
            cout << "[1] 프로세스를 찾았습니다 !! :D" << endl;
            DWORD targetProcess = GetWindowThreadProcessId(hWnd, &procID);
            if (targetProcess)
            {
                cout << "[2] PID를 가져오는데 성공했습니다 !! :D" << endl;
                hAndle = OpenProcess(PROCESS_ALL_ACCESS, 0, procID);
                BASE_ADDR = GetModuleBaseAddress(procID, L"ac_client.exe");
                PLAYER_BASE = BASE_ADDR + 0x0017E0A8;
                cout << "Target PID : " << procID << endl << "ModuleBaseAddr : " <<
BASE_ADDR << endl << "PlayerBase : " << PLAYER_BASE << endl;
                HotKeyActivate = true;
                break;
            }
        }
    }

    if (HotKeyActivate == true)
    {
        cout << "===== HOT KEYS =====" << endl;
        cout << "[NUMPAD 1] : HP 999" << " || " << "[NUMPAD 2] : 탄약무한" << endl;
        cout << "[NUMPAD 3] : 노딜레이" << " || " << "[NUMPAD 4] : 총기반동제거" << endl;
        while (true)
        {
            Sleep(100);
            if (GetAsyncKeyState(VK_NUMPAD1))
            {
                DWORD oldProtect;
                VirtualProtectEx(hAndle, (BYTE*)PLAYER_BASE, 1, PAGE_EXECUTE_READWRITE,
&oldProtect);
                ReadProcessMemory(hAndle, (BYTE*)PLAYER_BASE, &pAddress,
sizeof(pAddress), NULL);
                pAddress += 0xEC;
                newValue = 999;
                WriteProcessMemory(hAndle, (BYTE*)pAddress, &newValue, sizeof(newValue),
NULL);
                VirtualProtectEx(hAndle, (BYTE*)PLAYER_BASE, 1, oldProtect, &oldProtect);
            }
        }
    }
}
```

main.cpp

```
if (GetAsyncKeyState(VK_NUMPAD2))
{
    memory::HackActivate(hAndle, (BYTE*)(BASE_ADDR + 0xC73EF), (BYTE*)"\\x90
\x90", 2);
}

if (GetAsyncKeyState(VK_NUMPAD3))
{
    DWORD oldProtect;
    VirtualProtectEx(hAndle, (BYTE*)PLAYER_BASE, 1, PAGE_EXECUTE_READWRITE,
&oldProtect);
    ReadProcessMemory(hAndle, (BYTE*)PLAYER_BASE, &pAddress,
sizeof(pAddress), NULL);
    pAddress += 0x164;
    newValue = 0;
    WriteProcessMemory(hAndle, (BYTE*)pAddress, &newValue, sizeof(newValue),
NULL);
    VirtualProtectEx(hAndle, (BYTE*)PLAYER_BASE, 1, oldProtect,
&oldProtect);
}

if (GetAsyncKeyState(VK_NUMPAD4))
{
    memory::HackActivate(hAndle, (BYTE*)(BASE_ADDR + 0xC8BA0), (BYTE*)"\\xC2
\x08\x00", 3);
}
```

memory.cpp

```
#include "memory.h"

void memory::HackActivate(HANDLE hProc, BYTE* addr, BYTE* buf, unsigned int size)
{
    DWORD oldProtect;
    VirtualProtectEx(hProc, addr, size, PAGE_EXECUTE_READWRITE, &oldProtect);
    WriteProcessMemory(hProc, addr, buf, size, NULL);
    VirtualProtectEx(hProc, addr, size, oldProtect, &oldProtect);
}
```

memory.h

```
#include <Windows.h>

namespace memory
{
    void HackActivate(HANDLE hProc, BYTE* addr, BYTE* buf, unsigned int size);
}
```

offset.cpp

```
#include <TlHelp32.h>

using namespace std;

DWORD procID;
DWORD BASE_ADDR;
DWORD PLAYER_BASE;

HANDLE hHandle;

bool HotKeyActivate = false;

int pAddress;
int newValue;

DWORD GetModuleBaseAddress(DWORD pId, const wchar_t* moduleName)
{
    uintptr_t modBaseAddr = 0;
    HANDLE hSnap = CreateToolhelp32Snapshot(TH32CS_SNAPMODULE | TH32CS_SNAPMODULE32, pId);
    if (hSnap != INVALID_HANDLE_VALUE)
    {
        MODULEENTRY32 modEntry;
        modEntry.dwSize = sizeof(modEntry);
        if (Module32First(hSnap, &modEntry))
        {
            do
            {
                if (!_wcsicmp(modEntry.szModule, moduleName))
                {
                    modBaseAddr = (uintptr_t)modEntry.modBaseAddr;
                    break;
                }
            } while (Module32Next(hSnap, &modEntry));
        }
    }
    CloseHandle(hSnap);
    return modBaseAddr;
}
```