DOOM95 | Making an aimbot

Init Partners

Reverse Engineering

gamehacking

---

exploit (exploit) #1 March 16, 2020, 9:16pm

In the name of Allah, the most beneficent, the most merciful.

Introduction

“الأفكار تغير طابعك، أما الأفعال فتغير واقعك.”

I’ve played lots of classic games as a child, one that I particularly enjoyed was called **DOOM**, its concept was overly simple:

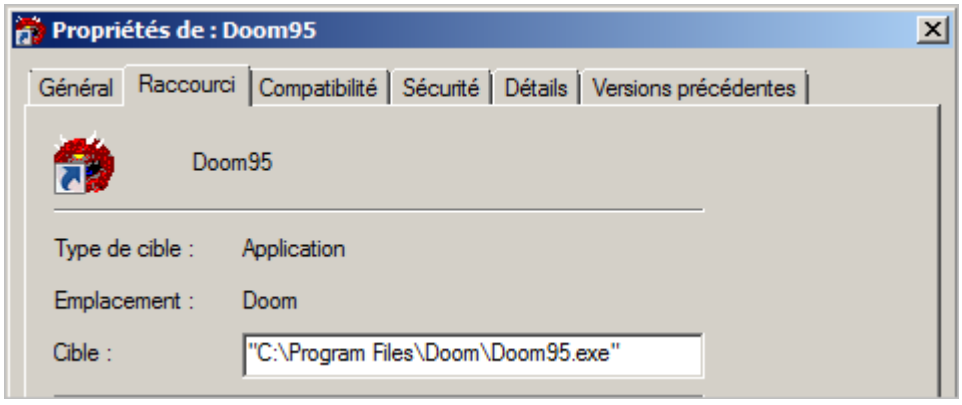
- Kill monsters that spawn all over the map. (%)
- Collect items. (%)
- Unlock each level’s secret.

But as the saying goes: *“There is beauty in simplicity”*.

Those days are *long gone*, and although everything that surrounds me changed, **I didn’t**.
I guess few things never *vanish*.
Note: I might do things wrong, but it’s all for fun anyway 🙂!

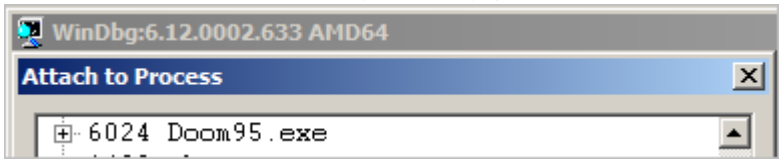
And so it all began

The shareware is available to download from ***ModDB***.

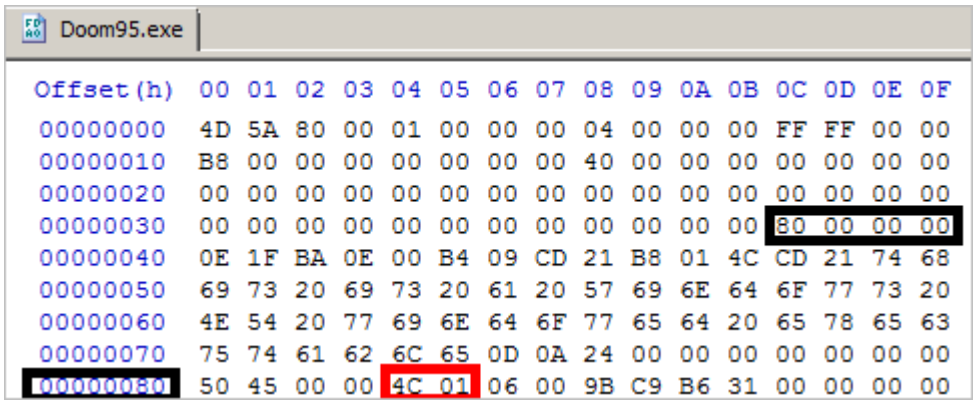


I started off by playing the game for a while, it reminded me of the implemented movement system.
The *left/right* arrow-keys allow **screen rotation**.
While *up/down* keys *render* **forward and backward moves** possible.

In order to look for the Image's entry point, I used WinDBG and attached to *Doom95.exe* process.



As you may have noticed, I'm running on a *64-bit machine*.
But the executable is 32-bit:



IMAGE_DOS_HEADER's *lfanew* holds the *Offset* to the *PE signature*.

—

COFF File Header (Object and Image)

At the beginning of an object file, or immediately after the signature of an image file, is a standard COFF file header in the following format. Note that the Windows loader limits the number of sections to 96.

Offset	Size	Field	Description
0	2	Machine	The number that identifies the type of target machine. For more information, see Machine Types.

The field next to **"PE\x00\x00"** is called *'Machine'*, a **USHORT** indicating *its type*.
so I proceeded to switch to *x86 mode* using **wow64exts**.

Command	Description
!wow64exts.sw	Switches between x86 and native mode.

—

```
0:011> !wow64exts.sw
Switched to 32bit mode
```

I then looked-up *"Doom"* within *loaded modules*, and used *$iment* to *extract* the *specified module*'s **entry point**.

Init **Partners**

```
0:011:x86> lm m Doom*
start             end                 module name
00400000 00690000   Doom95   C (no symbols)
10000000 10020000   Doomlnch C (export symbols)       Doomlnch.dll
0:011:x86> ? $iment(00400000)
Evaluate expression: 4474072 = 004444d8
```

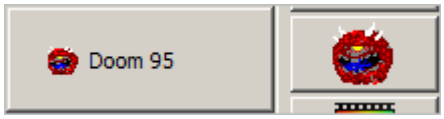
Jumping to that address in *IDA* reveals the function of interest: **_WinMain**.

```
BEGTEXT:004444D8                mov     ds:dword_618364, offset _WinMain
BEGTEXT:004444E2                jmp     loc_447477
```

The search for parts with beneficial information started.

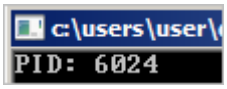
```
push    eax             ; nHeight
add     edx, ebx
push    edx             ; nWidth
push    0               ; Y
push    0               ; X
push    0x80CA0000      ; dwStyle
push    offset WindowName ; "Doom 95"
push    offset ClassName  ; "Doom95Class"
push    0x40000         ; dwExStyle
call    cs:CreateWindowExA
```

The **static** *WindowName* used by the call will result in our *fast retrieval* of *Doom's PID*.



The combination of **FindWindow()** and **GetWindowThreadProcessId()** makes this possible.

```
    HWND    DoomWindow;
    DWORD   PID;
    DoomWindow = FindWindow(NULL, _T("Doom 95"));

    if (! DoomWindow)
    {
        goto out;
    }

    GetWindowThreadProcessId(DoomWindow, &PID);
    printf("PID: %d\n", PID);

    out:
    return 0;
```



Init **Partners**

The next thing that caught my eye in the *_WinMain procedure* were the following lines.

```
loc_43AF74:
mov     edi, 1
mov     eax, ds:dword_60B00C
xor     ebp, ebp
mov     ds:dword_60B450, edi
mov     ds:dword_4775CC, ebp
cmp     eax, edi
jz      short loc_43AFB8
call    cs:GetCurrentThreadId
push    eax             ; dwThreadId
mov     edx, ds:hInstance
push    edx             ; hmod
push    offset fn       ; lpfn
push    2               ; idHook
call    cs:SetWindowsHookExA
mov     ds:hhk, eax
```

The function SetWindowsHookEx installs a **hook**(*fn*) within the *current thread* to *monitor System Events*. This example specifically uses an idHook that equals 2, which according to *MSDN* refers to **WH_KEYBOARD**.

WH_KEYBOARD 2	Installs a hook procedure that monitors keystroke messages. For more information, see the KeyboardProc hook procedure.

The callback function has the documented ***KeyboardProc*** prototype. It captures the ***Virtual-Key code*** in *wParam*.
On top of that, the *fn* function invokes *GetAsyncKeyState* to check if a *specific key is pressed* too:

VK_CONTROL 0x11	CTRL key
VK_MENU 0x12	ALT key

The function that handles *arrow-keys* is *sub_442A90*.

VK_LEFT 0x25	LEFT ARROW key
VK_UP 0x26	UP ARROW key
VK_RIGHT 0x27	RIGHT ARROW key
VK_DOWN 0x28	DOWN ARROW key

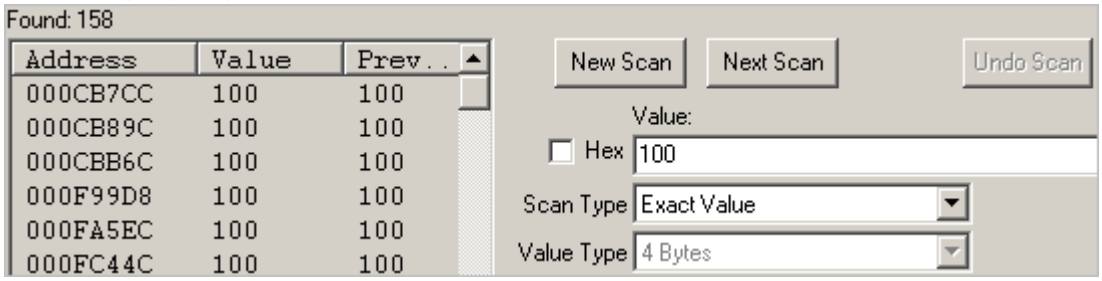
Init **Partners**

—

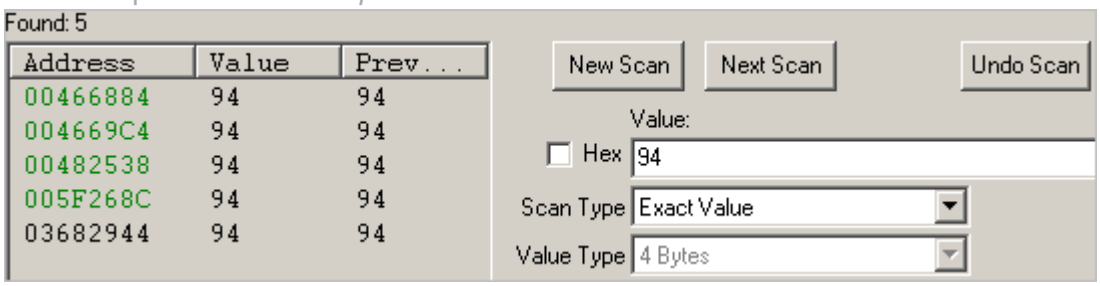
```
loc_439229:
mov     ebx, [esp+2Ch+v14]
mov     edx, [esp+2Ch+lParam]
mov     eax, esi
call    sub_442A90
```

Player information

Our ignorance of the structure stored within memory puts us at a disadvantage.
In order to find where *Health is mainly stored*, I'll be using *Cheat Engine* with the assumption that it is a *DWORD*(*4 bytes*): *"\x64\x00\x00\x00"*.



I'll then proceed to get the *character damaged*, and *'Next Scan'* for the *new value*.
We end up with **5** different *pointers*.

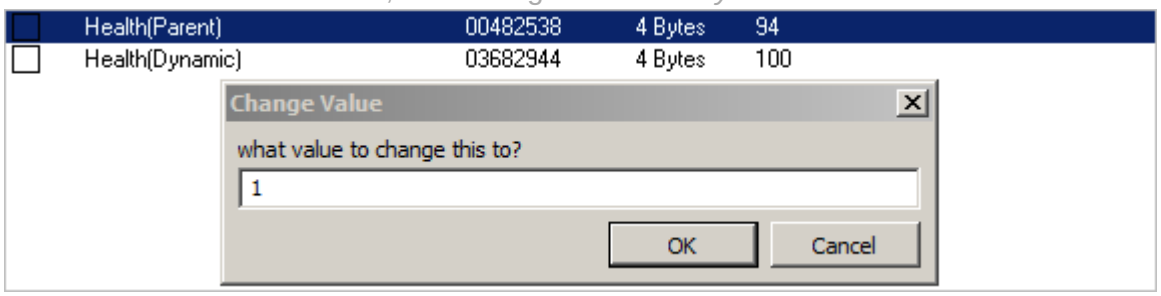


The **last one** is of a *black color*, meaning that it is a *dynamic address*, modifying it doesn't result in any **observable change**.
Others are clearly static, modifying *3/4* of them leads to *restoring the original value*, which meant that the *1/4* left is **the parent**, and the *rest just copy its value*.
00482538: Health that appear on the screen.
03682944: A promising address because it is *not updated with the parent*.

Health is stored in *two different locations*, which means one is *nothing but a decoy*.

Init **Partners**

I set the *first one's value to **1***, and then got *attacked by a monster*.

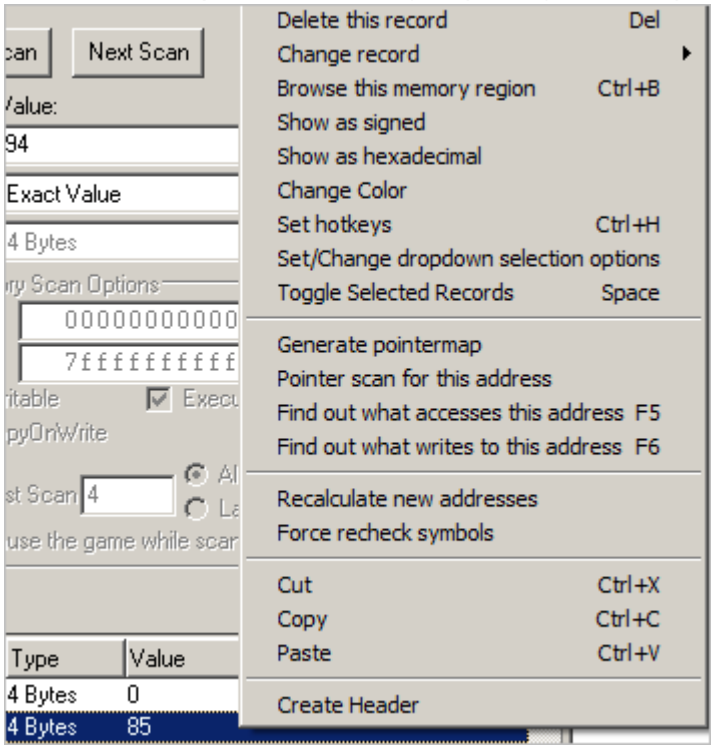


The result is:

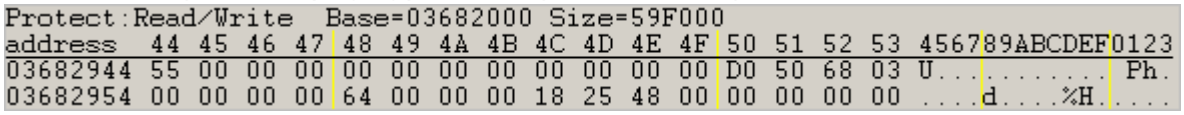
The value that *appears on the screen hangs at 0*, and the *character doesn't die*.

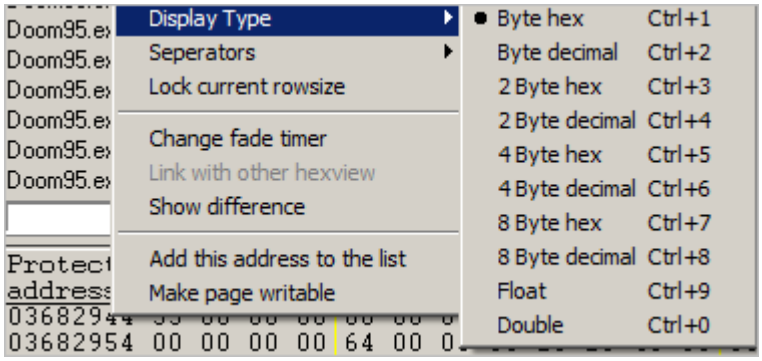
While the second kept *decreasing on each attack*, meaning that it effectively held *the real value*.

We need to *inspect the memory region* by selecting the value and clicking *CTRL+B*.



We can change the *Display Type* from **Byte hex** to **4 Byte hex**.





–

Protect:Read/Write Base=03682000 Size=59F000

address	44	48	4C	50	456789ABCDEF0123
03682944	00000055	00000000	00000000	036850D0	U.....Ph.
03682954	00000000	00000064	00482518	00000000	....d....%H.....

This is easier to work with, I *started searching* for the **closest pointer** in a *limited range*, and it turned out to be **3682A24**.

036828D4	03682344	00484CF8	03682A24	004250D0
036828E4	06A8AC64	F6B4558E	00000000	036BC418
036828F4	00000000	C2C00000	0000001C	00000000
03682904	00000000	00000000	036782F4	00000000
03682914	00E00000	00100000	00380000	00001FD8
03682924	00003AEB	00000000	00000000	00000000
03682934	00460B48	FFFFFFFF	0045B1D0	02000C46
03682944	00000055	00000000	00000000	036850D0

We then *goto address* to see its content:

03682A24	036828D8	03682AD8	004250D0	01200000
03682A34	F3E00000	00280000	00000000	00000000
03682A44	40000000	00000075	00000000	00000000
03682A54	00000000	03678684	00280000	00B80000
03682A64	00100000	00100000	00000000	00000000
03682A74	00000000	00000000	00000062	00462E80
03682A84	FFFFFFFF	004605B4	00000002	000003E8
03682A94	00000000	00000000	00000000	00000008

Notice that the *Object's health is empty*, and that *the struct* holds a **Backward and Forward link** at *its start*.

A spark

The idea that saved me a lot of time!
CHEAT CODES, I was both happy and shocked to find out that they really existed!

DOOMWiki includes messages that appear on detection of each message.
Two commands were exceptional because of the information they manipulate.

idmypos[note 10]	Shows the player's coordinates and compass direction. These numbers are hexadecimal representations of Doom's internal coordinate format, and can therefore be difficult to read; an online IDMYPOS converter can be used to decode them.	"ang=*{angle}*;x,y=(*{x}*,*{y}*)"

–

fhhall[1]	Kills all monsters in the level, excepting lost souls. (Pain elementals killed in this manner will still generate extra lost souls as normal).

1. HALL refers to Jason Hall, CEO of Monolith Productions, who requested that particular cheat (hence the "BY REQUEST..." message that displays when the code is executed). The actual damage done to each monster is 10,000 hit points.

The magical keywords: *"ang=" and "BY REQUEST…"*.
The first one's usage occurs in:

```
loc_432776:
mov     eax, offset off_4669BC
movsx   edx, byte ptr [ebp+4]
call    sub_414E50
test    eax, eax
jz      short loc_4327D4
mov     edx, ds:dword_482A7C
lea     eax, ds:0[edx*8]
add     eax, edx
lea     eax, ds:0[eax*4]
sub     eax, edx
mov     eax, ds:dword_482518[eax*8]
```

```
mov     ebx, [eax+10h]
push    ebx                                                    Init      Partners
mov     ecx, [eax+0Ch]
push    ecx
mov     esi, [eax+20h]
push    esi
push    offset aAng0xXXY0xX0xX ; "ang=0x%x;x,y=(0x%x,0x%x)"
push    offset unk_5F2758
call    sprintf_
```

This is important and worthy to be added to our *CE Table*.

☐	Index	00482A7C	4 Bytes	0
☐	PPlayer	00482518	4 Bytes	03A019F8

A struct layout is also to be concluded:
@) +0x10: y
@) + 0xC: x
@) +0x20: angle
I immediately noticed the missing *Z coordinate*.
I knew it existed, I mean, there's stairs. (Hey, don't laugh! 😟)
I ended up realizing that it is at *+0x14* after a few tests. (Up and down we go.)

I knew that even if the enemy is on a different altitude: The shot still hits, and so I ignored Z.
X, Y and Angle on the other hand are majorly important because of *distance calculation* and *angle measurement*.
The values they *held look weird*, are *they floats*?

☐	PPlayer	00482518	4 Bytes	036DA1E4
☐	Player	036DA1E4	4 Bytes	00484CF8
☐	X	036DA1F0	4 Bytes	041FEC4B
☐	Y	036DA1F4	4 Bytes	F296BCD2
☐	Z	036DA1F8	4 Bytes	00000000
☐	Angle	036DA204	4 Bytes	2AC00000

☐	Angle	036DA204	Float	3.410605132E-13

No, doesn't look like it. All I knew for now is that the *view changes upon modification*.

Moving on to the second:

```
loc_432533:
mov     eax, offset off_466898
movsx   edx, byte ptr [ebp+4]
call    sub_414E50
test    eax, eax
jz      short loc_43255E
mov     eax, ds:dword_5F274C
mov     dword ptr [eax+0D8h], offset aByRequest___ ; "By request..."
call    sub_420C50
jmp     loc_4326BA
```

We can see that it only passes execution to *sub_420C50*, and that's where the magic happens.

```
sub_420C50 proc near
push    ebx
push    ecx
push    edx
push    esi
mov     esi, ds:dword_484CFC
cmp     esi, offset dword_484CF8
jz      short loc_420C92
loc_420C62:
cmp     dword ptr [esi+8], offset sub_4250D0
jnz     short loc_420C87
test    byte ptr [esi+6Ah], 40h
jz      short loc_420C87
cmp     dword ptr [esi+6Ch], 0
jle     short loc_420C87
mov     ecx, 2710h
mov     eax, esi
xor     ebx, ebx
xor     edx, edx
call    sub_422370
loc_420C87:
mov     esi, [esi+4]
cmp     esi, offset dword_484CF8
jnz     short loc_420C62
```

We can see that it traverses a *list of Objects*, starting with **[484CFC]** and ending if the *Forward link(+4) equals* **484CF8**.

☐	PPlayer	00482518	4 Bytes	03A019F8
☐	List	00484CFC	4 Bytes	03A019F8

The inclusion of **Player Object** in *the list* indicates that it contains *all available* **Entities**.
The three checks there are:
[Entity + 0x08] == 0x4250D0
[Entity + 0x6A] & 0x40

[Entity + 0x6C] > 0

I was curious on what the **Player Object** held at those *Offsets*:

☐	Player+0x8	P->03A01A00	4 Bytes	004250D0
☐	Player+0x6A	P->03A01A62	Byte	00
☐	Player+0x6C	P->03A01A64	4 Bytes	00000026

@) + 0x8: *Function pointer(Pass)*.

@) +0x6A: *Byte(Error)*, *seems like IsMonster check*.

@) +0x6C: *Health(Pass)*.

A small mistake

"Did anyone do this before?", I wondered.

So I searched for:

> intext:"ang=0x%x;x,y=(0x%x,0x%x)" doom

And well, I found out that the *source code was available*. 😂

github.com › blob › master › linuxdoom-1.10 › st... ▾ Traduire cette page

doom/st_stuff.c at master · historicalsource/doom · GitHub

DOOM (1993) by id Software, Inc. Contribute to historicalsource/**doom** ... **doom**/linuxdoom-1.10/st_stuff.c ... sprintf(buf, "**ang=0x%x;x,y=(0x%x,0x%x)**",.

At first *I was mad*, because I spent about *3 to 4 days to get the results previously stated*. But, hey! I *needed more information anyway*, and this was an *easy road showing up*.

```
sprintf(buf, "ang=0x%x;x,y=(0x%x,0x%x)",
        players[consoleplayer].mo->angle,
        players[consoleplayer].mo->x,
        players[consoleplayer].mo->y);
```

—

```
void ST_initData(void)
{

    int         i;

    st_firsttime = true;
    plyr = &players[consoleplayer];
```

—

```
// main player in game
static player_t*        plyr;
```

So the structure we look for is defined in ***d_player.h***, the *interesting element's name* is **mo**.

```
//
// Extended player object info: player_t
//
typedef struct player_s
{
    mobj_t*     mo;
...
```

Its nature is *mobj_t*, declared in ***p_mobj.h***.

Init **Partners**

```
// Map Object definition.
typedef struct mobj_s
{
    // List: thinker links.
    thinker_t       thinker;

    // Info for drawing: position.
    fixed_t     x;
    fixed_t     y;
    fixed_t     z;

    // More list: links in sector (if needed)
    struct mobj_s*  snext;
    struct mobj_s*  sprev;

    //More drawing info: to determine current sprite.
    angle_t     angle;  // orientation
...
```

The size of ***thinker_t*** is: *sizeof(PVOID) * 3* = 4 * 3 = **12**.
Then comes *X, Y and Z* at *(0x0C, 0x10, 0x14)*.
Two pointers *@**0x18*** are ignored(*4 * 2* = **8**).
Angle is at *0x20*.

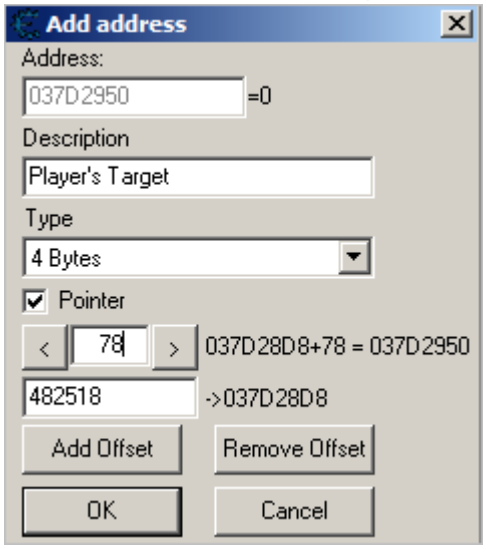
```
...
    int         health;

    // Movement direction, movement generation (zig-zagging).
    int         movedir;    // 0-7
    int         movecount;  // when 0, select a new dir

    // Thing being chased/attacked (or NULL),
    // also the originator for missiles.
    struct mobj_s*  target;
...
```

The target element is interesting, it supposedly holds a pointer to the *Map Object* being attacked!
Calculating its offset isn't that hard, because we know that *Health* is at *0x6C*.
*FIELD_OFFSET(mobj_t, target) = 0x6C + sizeof(int) * 3* = **0x78**.
The following line in ***r_local.h*** indicates that there's a **lookup table/function** for *Angles*, explaining why there's *weird values therein*.

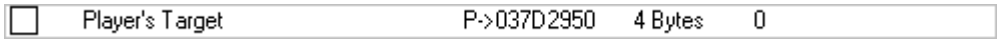
```
// Binary Angles, sine/cosine/atan lookups.
#include "tables.h"
```

Init **Partners**

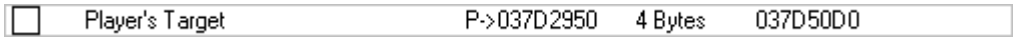
It's time to see what the *target* element holds for us!



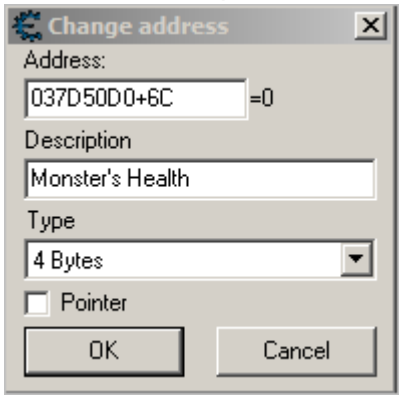
Since I just started up the game, its value is **NULL**.



Attacking or getting attacked by a monster leads to a *value change*.



But there is no *update after killing the monster, hmmm*.



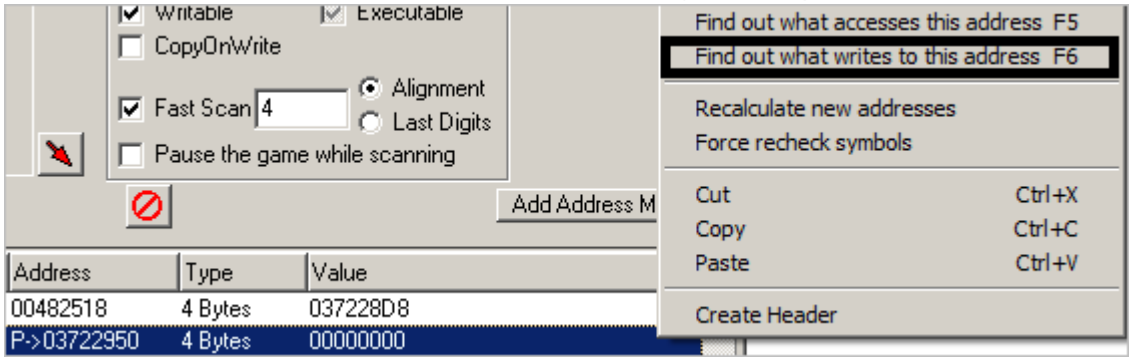
The **health** is the *only indicator of death if it is <= 0*.

And that's *not the only problem*:

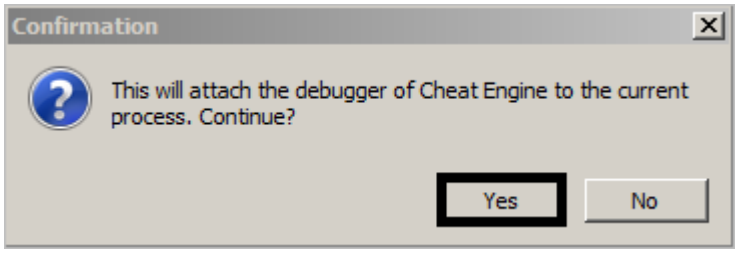
Attacking a second Monster doesn't result in any change occuring.

Since I want it to be *regularly updated*, I had to *find a way around it*.

I restarted *Doom95.exe*, selected the *Pointer to Player's Target* and:

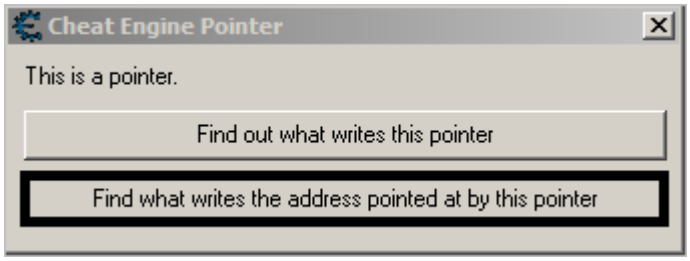


—

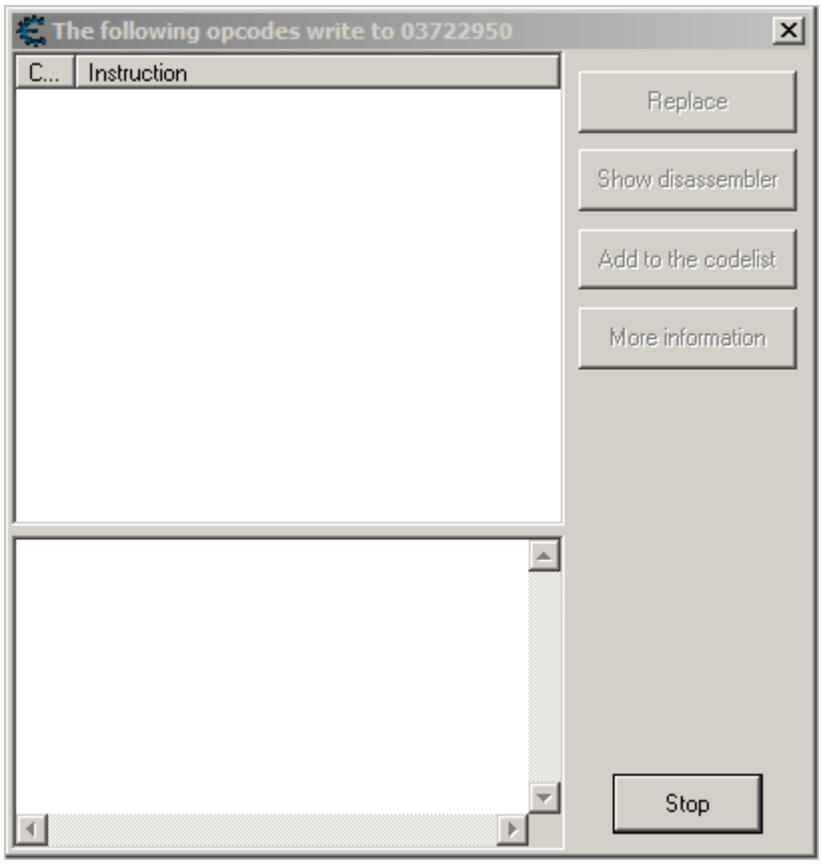


Init **Partners**

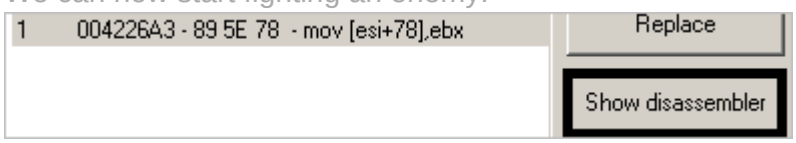
—



—



We can now start fighting an enemy:



This is the instruction *responsible for writing to the Player's Target element*.
Going back a little in *disassembly window*, there are some **simple checks**:
Is the Target NULL? Is it equal to the Player itself?

Init **Partners**

```
Doom95.exe+22 8B 5C 24 0C          mov   ebx,[esp+0C]
Doom95.exe+22 85 DB                test  ebx,ebx
Doom95.exe+22 74 49                je    Doom95.exe+226D5
Doom95.exe+22 39 DE                cmp   esi,ebx
Doom95.exe+22 74 45                je    Doom95.exe+226D5
Doom95.exe+22 83 7B 58 03          cmp   dword ptr [ebx+58],03   3
Doom95.exe+22 74 3F                je    Doom95.exe+226D5
Doom95.exe+22 C7 86 80000000 64000000  mov  [esi+00000080],00000064  100
Doom95.exe+22 8B 56 5C             mov   edx,[esi+5C]
Doom95.exe+22 89 5E 78             mov   [esi+78],ebx
```

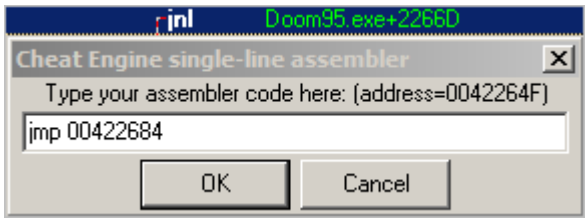
The *origin of EBX register* is the *selected instruction*, and *its location is*: **00422684**.

All I had to do is *find a location* where to place a **JMP 422684**.

I ended up choosing **0042264F**:

```
Doom95.exe+22 3B 42 20             cmp   eax,[edx+20]
Doom95.exe+22 7D 1C                jnl   Doom95.exe+2266D
Doom95.exe+22 F6 46 6B 01          test  byte ptr [esi+6B],01   1
Doom95.exe+22 75 16                jne   Doom95.exe+2266D
Doom95.exe+22 8A 4E 68             mov   cl,[esi+68]
Doom95.exe+22 80 C9 40             or    cl,40                  64
Doom95.exe+22 8B 56 5C             mov   edx,[esi+5C]
Doom95.exe+22 88 4E 68             mov   [esi+68],cl
Doom95.exe+22 89 F0                mov   eax,esi
Doom95.exe+22 8B 52 1C             mov   edx,[edx+1C]
Doom95.exe+22 E8 73240000          call  Doom95.exe+24AE0
Doom95.exe+22 8B 86 80000000       mov   eax,[esi+00000080]
Doom95.exe+22 C7 46 7C 00000000    mov   [esi+7C],00000000  0
Doom95.exe+22 85 C0                test  eax,eax
Doom95.exe+22 74 06                je    Doom95.exe+22684
Doom95.exe+22 83 7E 58 03          cmp   dword ptr [esi+58],03   3
```

–



–

```
Doom95.exe+22 EB 33                jmp   Doom95.exe+22684
```

The ***sequence of bytes*** *turns from {0x7D, 0x1C} to {0xEB, 0x33}*, we aren't *destroying any instructions after it*.

Let's now see if it *changes on each attack*:

Monster #1:

```
Player's Target   P->03722950   4 Bytes   037250D0
```

Monster #2:

```
Player's Target   P->03722950   4 Bytes   03722DA8
```

PERFECT!

Last piece of the puzzle

Monsters could *accurately aim at my character*.

I knew a function responsible for **angle measurement** existed, *I just had to find it*.

After a *few hours searching*, I ended up looking in ***p_enemy.c***;

```
boolean P_CheckMeleeRange (mobj_t*  actor)
{
```

```
    mobj_t* pl;
    fixed_t dist;

    if (!actor->target)
    return false;

    pl = actor->target;
    dist = P_AproxDistance (pl->x-actor->x, pl->y-actor->y);

    if (dist >= MELEERANGE-20*FRACUNIT+pl->info->radius)
    return false;

    if (! P_CheckSight (actor, actor->target) )
    return false;

    return true;
}
```

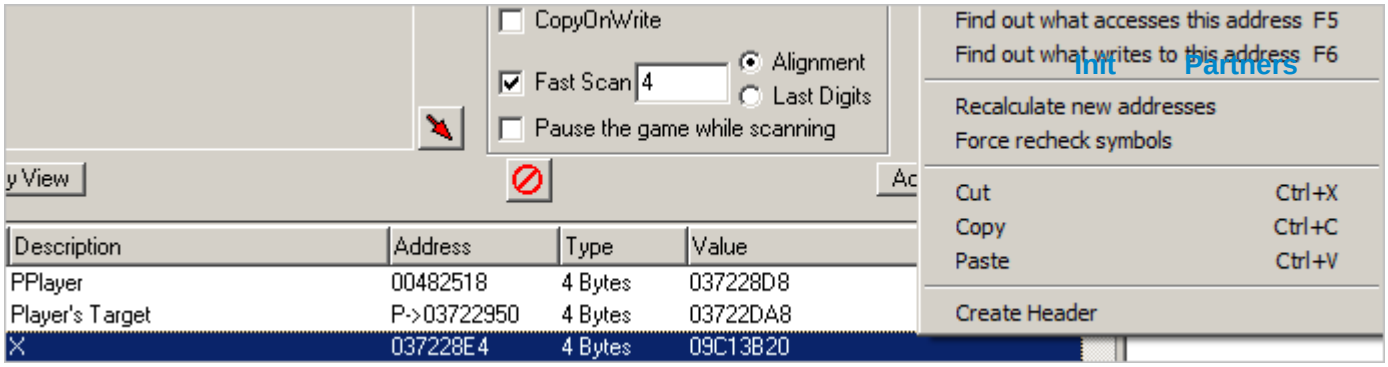

A collection of interesting functions!
P_AproxDistance()
P_CheckSight()
And the most **promising** one:

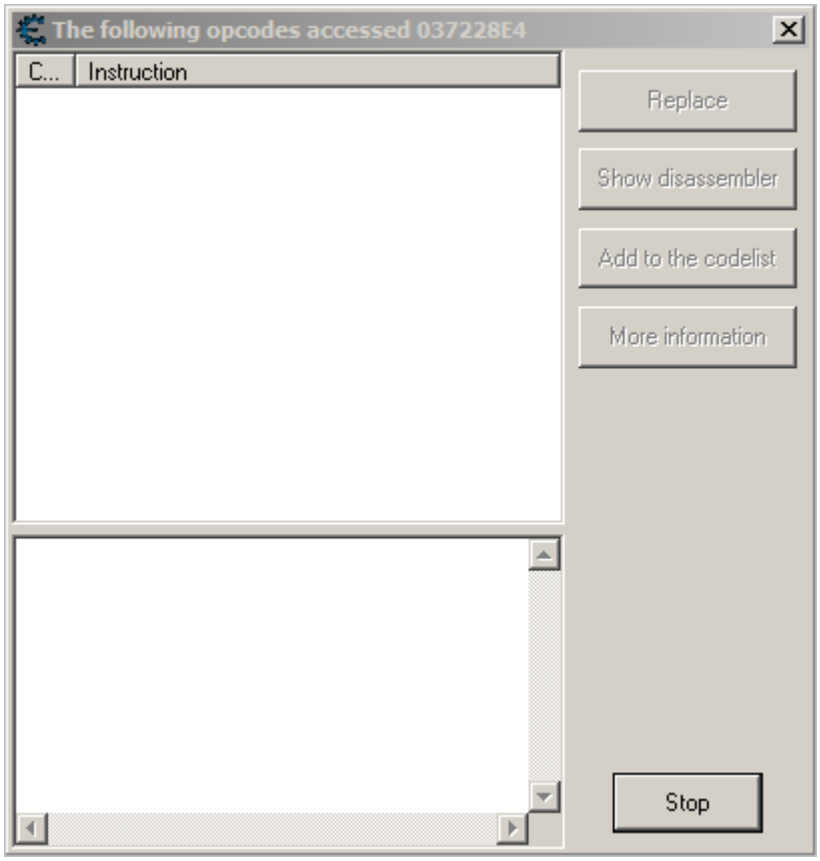
```
an = R_PointToAngle2 (actor->x,
                      actor->y,
                      player->mo->x,
                      player->mo->y)
    - actor->angle;
```

R_PointToAngle2(), and its ***definition*** is *the following*:

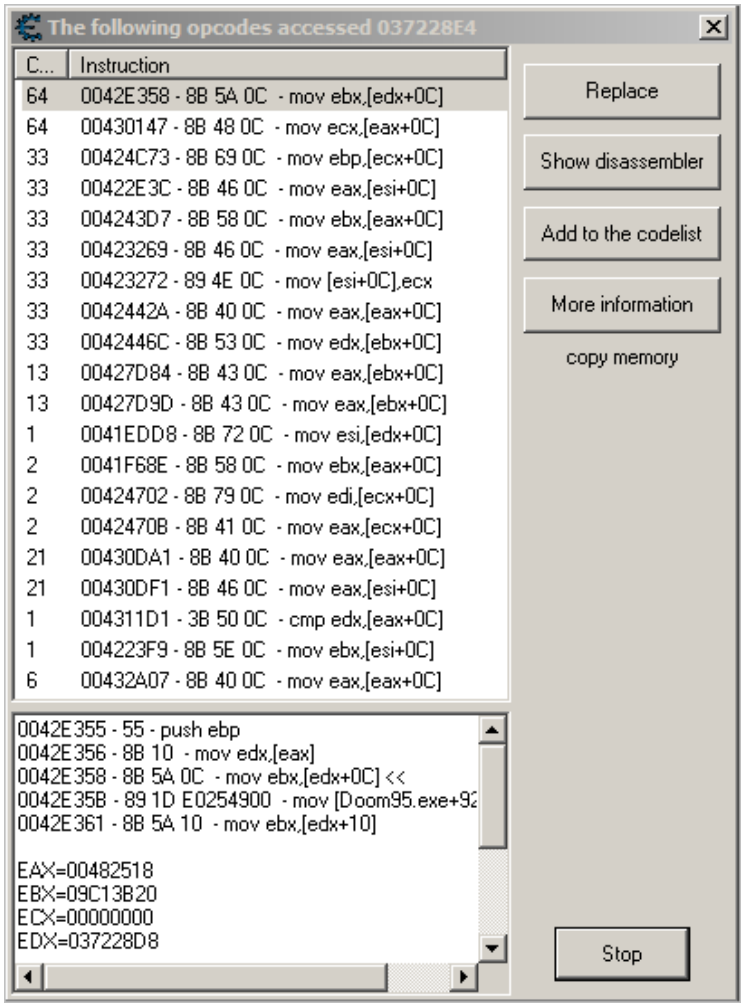
```
angle_t
R_PointToAngle2
( fixed_t   x1,
  fixed_t   y1,
  fixed_t   x2,
  fixed_t   y2 )
{
    viewx = x1;
    viewy = y1;

    return R_PointToAngle (x2, y2);
}
```

I knew the *Player's X, Y were read right before invokation*. I used this information to *trace the calls* and *watched for accesses*:





Once a *monster aims at us*, we **get results**:



Init **Partners**

I *started* with those with the *least hit-count at the middle*.

1	0041EDD8 - 8B 72 0C - mov esi,[edx+0C]
2	0041F68E - 8B 58 0C - mov ebx,[eax+0C]
2	00424702 - 8B 79 0C - mov edi,[ecx+0C]
2	0042470B - 8B 41 0C - mov eax,[ecx+0C]

The second one looks like the thing we're looking for!

```
mov  eax,[esi+78]
mov  edx,[esi+10]
mov  ecx,[eax+10]
mov  ebx,[eax+0C]
mov  eax,[esi+0C]
call Doom95.exe+2DB10
```

It prepares to call **0042DB10** by loading the *Target* in *EAX* and storing its *(X, Y) coordinates* in *EBX and ECX*, while *EAX and EDX* hold *those of the monster*.

We can deduce that it is a __fastcall.

Disassembling the function *shows*:

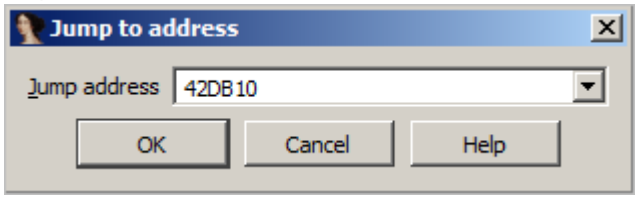
```
Doom95.exe+2D56          push  esi
Doom95.exe+2D57          push  edi
Doom95.exe+2D89 C6       mov   esi,eax
Doom95.exe+2D89 D7       mov   edi,edx
Doom95.exe+2D89 D8       mov   eax,ebx
Doom95.exe+2D89 CA       mov   edx,ecx
Doom95.exe+2D89 35 E0254900  mov  [Doom95.exe+925E0],esi  [09D74D80]
Doom95.exe+2D89 3D DC254900  mov  [Doom95.exe+925DC],edi  [F5A75666]
Doom95.exe+2DE8 B5FDFFFF call  Doom95.exe+2D8E0
Doom95.exe+2D5F          pop   edi
Doom95.exe+2D5E          pop   esi
Doom95.exe+2DC3          ret
```

Looks familiar!
It is R_PointToAngle2() @ ***0042DB10*** *!*

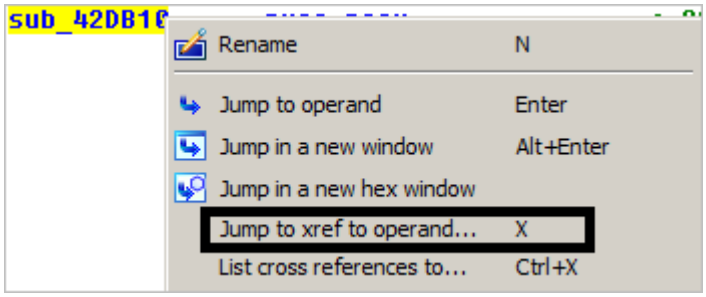
With that in *mind, locating this function is made easier*.

```
//
// A_FaceTarget
//
void A_FaceTarget (mobj_t* actor)
{
    if (!actor->target)
    return;

    actor->flags &= ~MF_AMBUSH;

    actor->angle = R_PointToAngle2 (actor->x,
                   actor->y,
                   actor->target->x,
                   actor->target->y);

    if (actor->target->flags & MF_SHADOW)
    actor->angle += (P_Random()-P_Random())<<21;
}
```

I'll just use *IDA*.



–



–

```
BEGTEXT:0041F670                push    ebx
BEGTEXT:0041F671                push    ecx
BEGTEXT:0041F672                push    edx
BEGTEXT:0041F673                push    esi
BEGTEXT:0041F674                mov     esi, eax
BEGTEXT:0041F676                cmp     dword ptr [eax+78h], 0
BEGTEXT:0041F67A                jz      short loc_41F6BE
BEGTEXT:0041F67C                mov     ah, [eax+68h]
BEGTEXT:0041F67F                and     ah, 0DFh
BEGTEXT:0041F682                mov     [esi+68h], ah
BEGTEXT:0041F685                mov     eax, [esi+78h]
BEGTEXT:0041F688                mov     edx, [esi+10h]
BEGTEXT:0041F68B                mov     ecx, [eax+10h]
BEGTEXT:0041F68E                mov     ebx, [eax+0Ch]
BEGTEXT:0041F691                mov     eax, [esi+0Ch]
BEGTEXT:0041F694                call    sub_42DB10
```

Init **Partners**

Looks like it, *it starts by returning if Target is **NULL***, then *ANDs* [*Monster+0x68*] with **0xDF**. What's sad, is that *I was looking at it since the beginning* in **CE**, welp 😂.
*A_FaceTarget is at **0041F670***.

The making

All that *we've learned about the game* will allow us to start *wrapping things* in *C++*.
Let's create *ADoom.h*:

```
#ifndef __ADOOM_H__
#define __ADOOM_H__

class ADoom {
public:
    ADoom(DWORD);
    ~ADoom();
private:
    HANDLE  DH;
};

#endif
```

And *ADoom.c*:

```
#include <cstdio>
#include <cstdlib>
#include <stdexcept>
#include <tchar.h>
#include <Windows.h>

#include "ADoom.h"

ADoom::ADoom(DWORD CPID)
{
    DH = OpenProcess(PROCESS_ALL_ACCESS, FALSE, CPID);
```

```
    if (DH != INVALID_HANDLE_VALUE)
    {
        return;
    }

    throw std::runtime_error("Can't open process!");
}

ADoom::~ADoom(){
    CloseHandle(DH);
}
```

Init Partners

I'll create functions that *read(**rM**)/write(**wM**) to the process memory* by extending *both the header and source file*.
We are going to use *two WINAPI calls for that purpose*: **ReadProcessMemory()** and **WriteProcessMemory()**.

```
BOOL ReadProcessMemory(
  HANDLE  hProcess,
  LPCVOID lpBaseAddress,
  LPVOID  lpBuffer,
  SIZE_T  nSize,
  SIZE_T  *lpNumberOfBytesRead
);
```

|||||

```
BOOL WriteProcessMemory(
  HANDLE  hProcess,
  LPVOID  lpBaseAddress,
  LPCVOID lpBuffer,
  SIZE_T  nSize,
  SIZE_T  *lpNumberOfBytesWritten
);
```

_

```
    template<typename ReadType>
    ReadType rM(DWORD, DWORD);
    BOOL wM(DWORD, PVOID, SIZE_T);
```

_

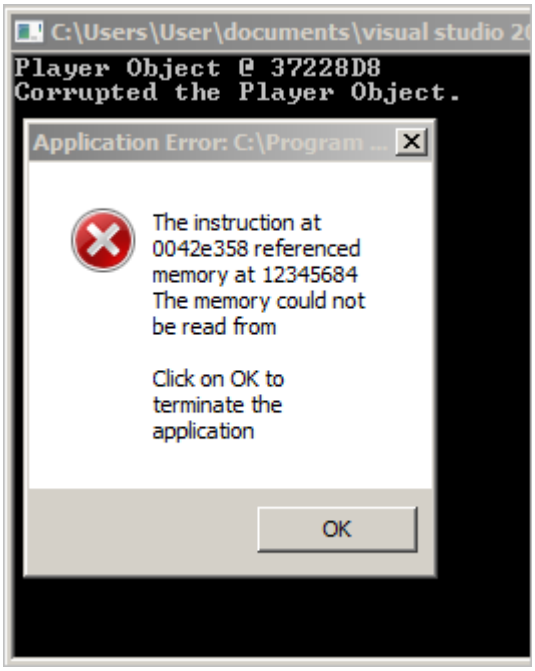
```
template<typename ReadType>
ReadType ADoom::rM(DWORD RAddress, DWORD Offset)
{
    ReadType Result;
    PVOID    External = reinterpret_cast<PVOID>(RAddress + Offset);

    ReadProcessMemory(DH, External, &Result, sizeof(Result), NULL);
    return Result;
}

BOOL ADoom::wM(DWORD RAddress, PVOID LAddress, SIZE_T Size)
{
    BOOL    Status = FALSE;
```

```
    PVOID   External = reinterpret_cast<PVOID>(RAddress);
                                                          Init      Partners
    if (WriteProcessMemory(DH, External, LAddress, Size, NULL))
    {
        Status = TRUE;
    }

    return Status;
}
```

Let's check if Player's Object manipulation is possible:

```
    try
    {
        ADoom   DAim(PID);
        DWORD   Corrupt = 0x12345678, Player, PPlayer = 0x482518;

        Player = DAim.rM<DWORD>(PPlayer, 0);
        printf("Player Object @ %lX\n", Player);

        DAim.wM(PPlayer, &Corrupt, sizeof(Corrupt));
        puts("Corrupted the Player Object.");
    } catch (const std::runtime_error &err) { }
```

_



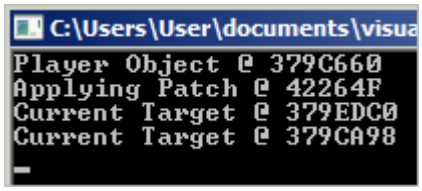
The ***Doom95.exe*** *process crashes*, *success*.
We have to *apply the* ***2 byte*** *patch and keep an eye* on the **Player's Target** value.

```
    try
    {
```

```
ADoom   DAim(PID);
BYTE    Patch[2] = {0xEB, 0x33};
DWORD   PAddress = 0x42264F;
DWORD   Player, PPlayer = 0x482518;
int     THealth;
DWORD   OTarget = 0, Target;

Player = DAim.rM<DWORD>(PPlayer, 0);
printf("Player Object @ %lX\n", Player);

printf("Applying Patch @ %lX\n", PAddress);
DAim.wM(PAddress, &Patch[0], sizeof(Patch));

while (true)
{
    Target = DAim.rM<DWORD>(Player, 0x78);

    // Are we currently engaging the enemy?
    if (Target != 0)
    {
        // If yes, is it already dead?
        THealth = DAim.rM<int>(Target, 0x6C);
```

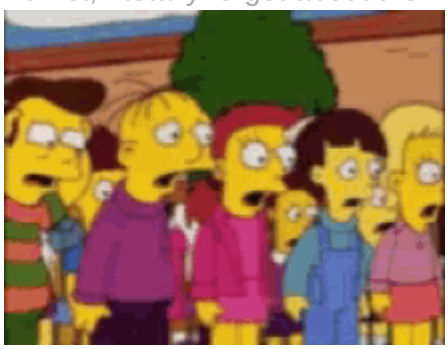
Init Partners

_



So far so good, we are making progress.

At first, I totally forgot about the **Health check**, and *it kept aiming at the dead Monster*.



It is time to use our knowledge about *A_FaceTarget(**0041F670**)*.

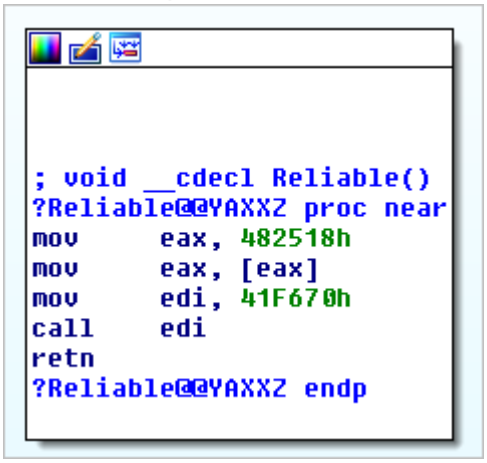
It takes an *mobj_t** argument in **EAX**, and *performs a single check*(*EAX->target != NULL*) before *calculating and storing the correct angle*, this is a *minimum of work on our side*.

All is left to do, is creating a **reliable function** and *storing/running* it in the *remote thread*.

```
VOID _declspec(naked) Reliable(VOID)
{
    __asm {
        mov eax, 0x482518 // Load PPlayer in EAX
```

```
        mov eax, [eax]    // Load Player Object in EAX
        mov edi, 0x41F670 // Indicate the FP(A_FaceTarget) Init    Partners
        call edi          // Call it
        ret
    }
}
```

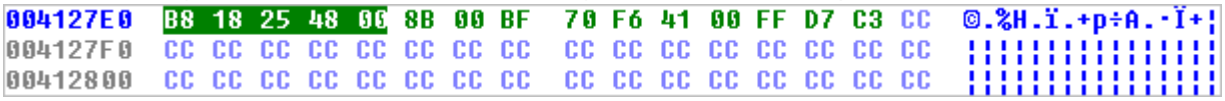
We can *compile the executable and load it up in IDA*.



```
; void __cdecl Reliable()
?Reliable@@YAXXZ proc near
mov     eax, 482518h
mov     eax, [eax]
mov     edi, 41F670h
call    edi
retn
?Reliable@@YAXXZ endp
```

_

Hex-view *is synchronized with* ***Disassembly-view****, so selecting the first 'mov' is all we have to do.*



```
004127E0  B8 18 25 48 00 8B 00 BF  70 F6 41 00 FF D7 C3 CC  ©.%H.ï.+p÷A.·Ï+¦
004127F0  CC CC CC CC CC CC CC CC  CC CC CC CC CC CC CC CC  ################
00412800  CC CC CC CC CC CC CC CC  CC CC CC CC CC CC CC CC  ################
```

That is *our function*!

```
BYTE Payload[] = {0xB8, 0x18, 0x25, 0x48, 0x00, 0x8B, 0x00,
                  0xBF, 0x70, 0xF6, 0x41, 0x00, 0xFF, 0xD7,
                  0xC3};
DWORD PSize = sizeof(Payload);
```

With that done, *we need a location to write it to*, it needs to be *Executable/Readable and Writeable too*.
In order to get it, we will call **VirtualAllocEx()**.

```
LPVOID VirtualAllocEx(
  HANDLE hProcess,
  LPVOID lpAddress,
  SIZE_T dwSize,
  DWORD  flAllocationType,
  DWORD  flProtect
);
```

We have to specify *flProtect* as *PAGE_EXECUTE_READWRITE*.
Another *helper function* will be called **aM** short for *allocate Memory*. 🙂

```
DWORD aM(SIZE_T);
```

_

```
DWORD ADoom::aM(SIZE_T Size)
{
    LPVOID RAddress = VirtualAllocEx(DH, NULL, Size, MEM_COMMIT | MEM_RESERVE
                      PAGE_EXECUTE_READWRITE);
    DWORD  Cast = reinterpret_cast<DWORD>(RAddress);

    return Cast;
}
```

Init Partners

And then there should be a *function to spawn a Thread* in *Doom95.exe process*.
We'll be using **CreateRemoteThread()**, and *wait for it to terminate* using **WaitForSingleObject()**.

```
HANDLE CreateRemoteThread(
  HANDLE                 hProcess,
  LPSECURITY_ATTRIBUTES  lpThreadAttributes,
  SIZE_T                 dwStackSize,
  LPTHREAD_START_ROUTINE lpStartAddress,
  LPVOID                 lpParameter,
  DWORD                  dwCreationFlags,
  LPDWORD                lpThreadId
);
```

||

```
DWORD WaitForSingleObject(
  HANDLE hHandle,
  DWORD  dwMilliseconds
);
```

It'll be called **sT**.

```
VOID sT(DWORD);
```

_

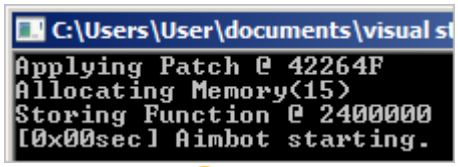
```
VOID ADoom::sT(DWORD FPtr)
{
    HANDLE RT;

    RT = CreateRemoteThread(DH, NULL, 0, (LPTHREAD_START_ROUTINE) FPtr,
         NULL, 0, NULL);

    if (RT != INVALID_HANDLE_VALUE)
    {
        WaitForSingleObject(RT, INFINITE);
    }
}
```

That's all we need, *now we can implement the whole loop*:

```
    try
    {
        ADoom   DAim(PID);
        BYTE    Patch[2] = {0xEB, 0x33};
```

```
DWORD   PAddress = 0x42264F;
BYTE    Payload[] = {0xB8, 0x18, 0x25, 0x48, 0x00, 0x8B, 0x40,
                     0xBF, 0x70, 0xF6, 0x41, 0x00, 0xFF, 0xD7,
                     0xC3};
DWORD   Location, PSize = sizeof(Payload);
DWORD   PPlayer = 0x482518, Player, Target;
int     THealth;

/*
    Patch:
    An unconditional JMP instruction that allows Player->target
    to be updated on every attack.
*/
printf("Applying Patch @ %lX\n", PAddress);
DAim.wM(PAddress, Patch, sizeof(Patch));

printf("Allocating Memory(%d)\n", PSize);
Location = DAim.aM(PSize);

printf("Storing Function @ %lX\n", Location);
```


–



And it works! 😃

End

It took many attempts to get to the final product, but it certainly was fun!
I could not include *pictures or GIFs from the game because I didn't find a way to do it*, for that, I apologize.
Lots of modifications were made to guarantee *reliability*, an example would be *the Player object* is updated on *two events*: **Death/Level Change**.
And I also *got rid of some functions such as*:

```
VOID GetMonsters(vector<DWORD> *M, HANDLE Proc)
{
    DWORD   First = 0x484CFC, Last = 0x484CF8;
    int     MHealth;
    UCHAR   IsMonster;

    First = rM<DWORD>(First, Proc);
    Last = rM<DWORD>(Last, Proc);

    do {
        IsMonster = rM<UCHAR>(First + 0x6A, Proc);
```

```
        MHealth = rM<int>(First + 0x6C, Proc);
```

Init **Partners**

```
        // Is it a monster and is it alive?
        if ((IsMonster & 0x40) && (MHealth > 0))
        {
            M->push_back(First);
        }
    } while ((First = rM<DWORD>(First + 4, Proc)) != Last);
}
```

I didn't even need to include <cmath> in the end!
NIAHAHAHA!
~ exploit

17 Likes

---

pry0cc (Leader & Offsec Engineer & Forum Daddy) #2 March 16, 2020, 9:00pm

Sick article dude!

This is really creative stuff, you always kill it with your articles, keep it up man 🙂

You're going to go so far in this world, there are few people like you.

2 Likes

---

exploit (exploit) #3 March 16, 2020, 9:03pm

Thank you so much, I'll do my best ❤️!
I'm really happy I found the time to write this 😄!

2 Likes

---

ricksanchez #4 March 17, 2020, 5:31pm

Good stuff as always **@exploit** ! Really enjoyed the write-up 🙂 .
About time you wrote a new article 😛

2 Likes

---

exploit (exploit) #5 March 17, 2020, 5:53pm

I know, it took forever **@ricksanchez** . 😂
And thank youu, really happy you liked it! 😆

Init **Partners**

1 Like

Danus #8 March 19, 2020, 3:40pm

This is some pretty fucking cool stuff **@exploit** , good job!

3 Likes

---

exploit (exploit) #9 March 19, 2020, 4:44pm

Thank youu **@Danus** ! 😄❤️

1 Like

DamaneDz (DamaneDz) #10 March 20, 2020, 12:47pm

Nice one kho 😛❤️
One of the best gamehacking sheet I've *ever read* !

1 Like