1 Getting started

We execute GeoMe.exe to get a glimpse of the crackme. It looks like a regular native window with ownerdrawn content. By pressing the left mouse button a vertex is added to the canvas. If there are already other vertices the new vertex will be connected with the previous one. By pressing the right mouse button the drawn polygon is validated and there is a good boy/bad boy message for valid/invalid polygons.

PEiD only reports MASM32 / TASM32 so we can load the application straight in IDA Pro. Following the flow at the entry point we immediately recognize the structure of a native windows application. A window class is registered (.text:0040109C), a window is created (.text:004010E8) and all window messages are handled (.text:00401102). Once WM_QUIT is received the application returns. Hence the main crackme functions are located in the WndProc of the main window (.text:00401130).

2 Inside WndProc

The coding of the WndProc is pretty clean. WM_CREATE (.text:0040113), WM_DESTROY (.text:0040114E), WM_COMMAND (.text:0040116B), WM_LBUTTONDOWN (.text:004011A3), WM_RBUTTONDOWN (.text:00401274), WM_SIZE (.text:00401335) and WM_PAINT (.text:004013E2) are handled.

In the following subsections we will have a deeper look at the four most important messages (WM_PAINT, WM_SIZE, WM_LBUTTONDOWN, WM_RBUTTONDOWN) and how they are handled.

2.1 WM_PAINT

shows the most important part of the WM_PAINT handler. The number loaded in ecx contains the number of the vertices of the drawn polygon. That is why I name this memory location dwNumVertices. The location .data:0040341C contains the vertices itself, stored as POINT’s. I name this location aVertices.
2.2 WM_SIZE

shows the most important part of the WM_SIZE handler. The loword of the lParam (the new width) and the hiword of the lParam (the new height) are loaded into eax and ecx. Next, their product is divided with four, stored (I name this memory location dwWindowSize), divided with two and stored again (I name this memory location dwWindowSize_2).

The remaining code just stretches the x- and y-attributes of the vertices in aVertices according to the new window size. There is no need for me to explain this process as we do not need to understand it for solving the crackme.

2.3 WM_LBUTTONDOWN

shows the most important part of the WM_LBUTTONDOWN handler. If there are 32 vertices, a WM_RBUTTONDOWN message will be sent, otherwise the vertex will be added. We now know that there are 32 vertices at most, so we set the number of elements of aVertices to 32.

2.4 WM_RBUTTONDOWN

shows the most important parts of the WM_RBUTTONDOWN handler. If dwNumVertices is below or equal to 2 we will get the BadBoyMessage.

The following code connects the last vertex we added to list with the first one (a simple LineTo call).

Finally, the application ends up in calling a function (I name this one Check). If the return value (__stdcall) is not 0 then the wanted good boy message will be displayed. In a real life application, if we wanted to crack this application, it might be enough to patch the jnz with a jmp, but, as we are interested in the algorithm itself, we will have a deeper look at Check and check how deep the rabbit-hole goes...

3 Inside Check

Check starts right away with a call to another function. As this function updates various internal values I name this function Update.

3.1 Inside Update

The first part of this function (.text:00401507-.text:00401544) searches the vertex which has the lowest y-value. If there are several vertices which all have the same (the lowest) y-value, the vertex with the highest x-value will be taken. The index of the found vertex is stored in edx.
Next an array is filled with the numbers from 0 to dwNumVertices-1. It is obvious that this array contains indices, so I name it adwIndices. The number of elements is the number of elements of aPoints - 32. After filling the array with the indices the first index (0) is swapped with the index at edx (index of the topmost vertex).

Then dwNumVertices-1 is loaded into edx and adwIndices[1] is loaded into ebx. The function at .text:004015E1 is called. I name this function SortAdwIndices as it sorts the indices array in ebx in clockwise order. This function internally uses the function at .text:004016A2. I name this one CompareVertices. This functions determines whether two vertices (specified by their indices) and aPoints[adwIndices[0]] are clockwise by checking the sign of the two dimensional cross product (right-hand rule).

There is one slight problem with SortAdwIndices. How would the function go with the vertices in 8? Yes, there is no real way to order it fully clockwise. The next part of Update (.text:00401577-.text:004015DF - 10) detects those anticlockwise vertices in the SortAdwIndices array and saves the valid ones in the array at the memory location at .data:004032EE and the number of elements in the memory location .data:004032EA. I call the array adwValidIndices (its size must be equal to the size of adwIndices) and the memory location for the number of elements dwNumValidIndices. Internally the new ordering is pretty easy. Again the cross product is used (check the function at .text:00401643 - 9). The first two vertices are pushed on the stack and two counter (ecx and dwNumValidIndices) are set to 2. Then every vertex (aPoints[adwIndices[2]] ... aPoints[adwIndices[dwNumVertices-1]]) is checked whether it is clockwise with the two vertices whose indices are on the top of the stack. If they are clockwise, the index of the checked vertex will be pushed on the stack and the counters will be increased. If the vertices are not clockwise, the vertex on top of the stack will be popped and the loop will be executed again without any changes to the counters. Finally, once ecx reaches dwNumVertices, ecx vertices are popped back in adwValidIndices.

The algorithm implemented in Update is referred to as GRAHAM SCAN. This algorithm computes the convex hull of a specific set of vertices. That means, adwValidIndices stores the indices of the vertices that inclose all vertices in aVertices.

### 3.2 The final validation...

#### 3.2.1 The first requirement

The first requirement for a valid polygon can be seen in (dwNumVertices*(dwNumVertices-1)+2)&7 must be equal to dwNumValidIndices.
3.2.2 The second requirement

The second requirement for a valid polygon can be seen in 12 and 13. The surface area of the polygon specified by the vertices of \texttt{adwValidIndices} must be between \texttt{dwWindowSize_2} and \texttt{dwWindowSize}.

4 Drawing a valid polygon

As there are only these two requirements, drawing a valid polygon is simple. To meet the first requirement we just have to draw a polygon whose number of vertices that enclose all vertices (\texttt{dwNumValidVertices}), combined with the total number of vertices (\texttt{dwNumVertices}), is valid (2).

Meeting the second requirement is simple as well. We just have to draw our polygon so big or small that the surface area of the convex hull ranges of our polygon between \texttt{dwWindowSize_2} and \texttt{dwWindowSize}.

14 shows a valid polygon. The first vertex is the left one. There are seven vertices (\texttt{dwNumVertices==7}), four of them specify the convex hull (\texttt{dwNumValidVertices==4}). The content of \texttt{adwIndices} can be seen in 3 the content of \texttt{adwValidIndices} can be seen in 4.

5 Conclusion

This crackme is definitely a simple one. There is no anti-debugger code, the code is not obfuscated and the algorithms used are well-known and simple to identify. Nevertheless this crackme is one of the unusual crackmes and solving it was fun. I certainly hope you find this article a comprehensive and useful source of information.
Figure 1: WM_PAINT

```
.mtext:00401340 mov eax, [ebp+1Param]
.mtext:00401343 and eax, OFFFFh
.mtext:00401348 mov ebx, [ebp+1Param]
.mtext:0040134B shr ebx, 10h
.mtext:0040134E mul ebx
.mtext:00401350 shr eax, 2
.mtext:00401353 mov dwWindowSize, eax ; Width*Height/4
.mtext:00401358 shr eax, 1
.mtext:0040135A mov dwWindowSize_2, eax ; Width*Height/8
```

Figure 2: WM_SIZE
Figure 3: WM_LBUTTONDOWN

Figure 4: WM_RBUTTONDOWN

Figure 5: Update: .text:00401544-.text:00401565
on return: eax==1: clockwise
eax==0: cross product=0 & distance of IndexA and IndexB
to adwIndices[0] is equal
eax==-1: anticlockwise
Attributes: bp-based frame

int __stdcall CompareVertices(int dwIndexA, int dwIndexB)

CompareVertices proc near

; CODE XREF: SortAdwIndices+10; SortAdwIndices+29

CompareVertices proc near

CompareVertices proc near

; Figure 6: CompareVertices
void SortAdwIndices(DWORD* pdwIndices, DWORD dwNumElementsToSort)
{
    if(dwNumElementsToSort<=1)
    {
        return;
    }

    DWORD dw=dwNumElementsToSort-1;
    DWORD dw2=0;
    for(;;)
    {
        //is clockwise?
        if(CompareVertices(pdwIndices[0], pdwIndices[dw])>0)
        {
            --dw;
            continue;
        }
        for(;;)
        {
            if(dw==dw2)
            {
                std::swap(pdwIndices[0], pdwIndices[dw]);
                SortAdwIndices(pdwIndices, dw);
                SortAdwIndices(&pdwIndices[dw+1], dwNumElementsToSort-(dw+1));
                return;
            }
            ++dw2;
            //is anticlockwise?
            if(CompareVertices(pdwIndices[0], pdwIndices[dw2])<=0)
            {
                std::swap(pdwIndices[dw], pdwIndices[dw2]);
                break;
            }
        }
    }
}

Figure 7: C++ implementation of SortAdwIndices
Figure 8: How would SortAdwIndices go with these vertices?

```c
int __stdcall IsClockwise(int dwBaseA, int dwIndexB, int dwIndexC)
```

Figure 9: C++ declaration of IsClockwise
.text:00401577 push adwIndices
.text:0040157D push adwIndices+4
.text:00401583 mov ecx, 2
.text:00401588 mov dwNumValidIndices, 2
.text:00401592 jmp short loc_4015C6
.text:00401594 ; ---------------------------------------------------------------
.text:00401594 loc_401594: ; CODE XREF: Update+CB
.text:00401594 push ecx
.text:00401595 mov ecx, adwIndices[ecx*4]
.text:0040159C push ecx ; dwIndexC
.text:0040159D push dword ptr [esp+8] ; dwIndexB
.text:004015A1 push dword ptr [esp+10h] ; dwBaseA
.text:004015A5 call IsClockwise
.text:004015AA pop ecx
.text:004015AB or eax, eax
.text:004015AD jz short loc_4015BF
.text:004015AF push adwIndices[ecx*4]
.text:004015B6 inc ecx
.text:004015B7 inc dwNumValidIndices
.text:004015BD jmp short loc_4015C6
.text:00401594 ; ---------------------------------------------------------------
.text:004015BF loc_4015BF: ; CODE XREF: Update+AC
.text:004015BF pop edx
.text:004015C0 dec dwNumValidIndices
.text:004015C6 loc_4015C6: ; CODE XREF: Update+91d
.text:004015C6 dec edx
.text:004015C6 loc_4015C6: ; CODE XREF: Update+91d
.text:004015C6 ; Update+BC
.text:004015C6 cmp ecx, dwNumVertices
.text:004015CC jb short loc_401594
.text:004015CE mov ecx, dwNumValidIndices
.text:004015D4 loc_4015D4: ; CODE XREF: Update+DC
.text:004015D4 dec ecx
.text:004015D4 pop edx
.text:004015D6 mov adwValidIndices[ecx*4], edx
.text:004015DD jnz short loc_4015D4

Figure 10: Check: .text:00401577-.text:004015DF
Figure 11: The first requirement for a valid polygon:
Figure 12: The second requirement for a valid polygon:

```asm
.globl loc_401480
.loc_401480:
    mov eax, dwNumValidIndices
    cmp edx, ecx
    jnz BadBoy
    fldz
    dec ecx
    
.loc_401480: ; CODE XREF: Check+5C
    mov esi, dwNumValidIndices[ecx*4]
    mov edi, adwValidIndices[ecx*4]
    fild aVertices.x[esi*8]
    fisub aVertices.x[edi*8]
    fild aVertices.y[esi*8]
    fiadd aVertices.y[edi*8]
    fmulp st(1), st
    faddp st(1), st
    dec ecx
    jnz short loc_401480
    mov ecx, dwNumValidIndices
    dec ecx
    mov esi, adwValidIndices[ecx*4]
    mov edi, adwValidIndices
    fild aVertices.x[esi*8]
    fisub aVertices.x[edi*8]
    fild aVertices.y[esi*8]
    fiadd aVertices.y[edi*8]
    fmulp st(1), st
    faddp st(1), st
    fidiv dword_4033F6
    fild dwWindowSize_2 ; Width*Height/8
    fild dwWindowSize ; Width*Height/4
    fcomip st, st(2)
    jbe short BadBoy ; dwWindowSize<=st(2)
    fcomip st, st(1)
    jnb short BadBoy ; st(1)>=dwWindowSize_2
```

\[ a_{\text{x}} \] = \frac{a_{\text{x}} + b_{\text{x}}}{c_{\text{y}}} \quad \text{and} \quad \frac{a_{\text{y}} + b_{\text{y}}}{c_{\text{y}}} \]

- Figure 12: The second requirement for a valid polygon:
float f=0.0
for(int i=0;i<dwNumValidIndices-1;++i)
{
    f+=(aPoints[adwValidIndices[i]].x-aPoints[adwValidIndices[i+1]].x)*
        (aPoints[adwValidIndices[i]].y+aPoints[adwValidIndices[i+1]].y);
}
f+=(aPoints[adwValidIndices[dwNumValidIndices-1]].x-aPoints[adwValidIndices[0]].x)*
    (aPoints[adwValidIndices[dwNumValidIndices-1]].y+aPoints[adwValidIndices[0]].y);
f/=2.0;
if(dwWindowsSize_2<f && //dwWindowSize_2==Width*Height/8
    f<dwWindows) //dwWindowSize==Width*Height/4
{
    //Good boy
}
else
{
    //Bad boy
}

Figure 13: C++ pseudo implementation of the second requirement for a valid polygon
Figure 14: Good boy
Table 1: The result of the call to `SortAdwIndices` with the vertices seen in 8

<table>
<thead>
<tr>
<th>adwIndices[i]</th>
<th>0 1 2 4 3 5</th>
</tr>
</thead>
</table>

Table 2: `dwNumVertices` and `dwNumValidIndices` combinations which meet the first requirement

<table>
<thead>
<tr>
<th>dwNumVertices</th>
<th>dwNumValidIndices</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>10</td>
<td>4</td>
</tr>
<tr>
<td>12</td>
<td>6</td>
</tr>
<tr>
<td>13</td>
<td>6</td>
</tr>
<tr>
<td>15</td>
<td>4</td>
</tr>
<tr>
<td>18</td>
<td>4</td>
</tr>
<tr>
<td>20</td>
<td>6</td>
</tr>
<tr>
<td>21</td>
<td>6</td>
</tr>
<tr>
<td>23</td>
<td>4</td>
</tr>
<tr>
<td>26</td>
<td>4</td>
</tr>
<tr>
<td>28</td>
<td>6</td>
</tr>
<tr>
<td>29</td>
<td>6</td>
</tr>
<tr>
<td>31</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 3: `adwIndices` of the polygon seen in 14

<table>
<thead>
<tr>
<th>i</th>
<th>0 1 2 3 4 5 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>adwIndices[i]</td>
<td>0 6 5 4 2 3 1</td>
</tr>
</tbody>
</table>
Table 4: \textit{adwValidIndices} of the polygon seen in [14]

<table>
<thead>
<tr>
<th>i</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>adwValidIndices[i]</td>
<td>0</td>
<td>6</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>