Contents

Introduction

A "code-what"? Unless you have spent some time working in the area of reverse engineering, chances are you have not heard of the term "codecave" before. If you have heard of it, you might not have read a clear definition of it or quite understand what it is or why it is useful. I have even asked seasoned assembly programmers about the term before and most of them had not heard of it. If it is new to you, do not worry, you are not the only one. It is a term that is scarcely used and is only useful in a reverse engineering context. Furthermore, is it "codecave" or "code cave"? I am not quite sure, but I will try my best to refer to it consistently as a "codecave". A space may sneak in there from time to time.

If you search around on the internet, you will not find much on the topic of codecaves. If you do, most of the resources are found on "shady" sites. It is true that codecaves have an important and useful place in the underground world of hacking, but they can be used for legitimate reasons as well (as with anything programming related that can be used for good or bad). Regardless, codecaves are just another tool a programmer or reverse engineer can use to enhance their skills and toolset. You may not have an immediate use for it now, but perhaps one day you will and you will be glad that you know how to use the concept.

The purpose of this article is to provide a complete guide to understanding and using codecaves. By the end of this article, you will know what a codecave is, what it is useful for, and how to use it. In addition, you will be exposed to a practical example to reinforce what you have learnt, so you can see the concept in action. This article is written as a guide for all levels of expertise,

even beginners, but it is assumed that you have some basic knowledge of C/C++, Assembly, and Reverse Engineering concepts. As you read along in the article, you might want to search the net for additional reference material if something is not quite clear.

This article is broken down into four main sections with various subsections. The "Introduction", what you are reading now, will setup the article and cover what this article is about and what you will hopefully learn. The "Theory" will discuss the theory of codecaves in regards to what they are and how they are used. The "Application" will show the "Theory" section in action with a complete example of using codecaves to accomplish a particular task. Last but not least, the "Conclusion" will quickly recap what the article has discussed and present parting words.

Now that the boring stuff is out of the way, it is time to get started!

Theory

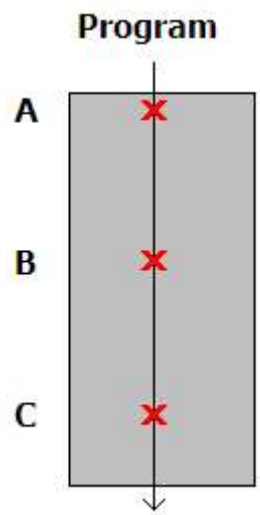
A codecave can best be defined as "*a redirection of program execution to another location and then returning back to the area where program execution had previously left.*" In a sense, a codecave is no different in concept than a function call, except for a few minor differences. If a codecave and a function call are so similar, why do we need codecaves at all then? The reason we need codecaves is because source code is rarely available to modify any given program. As a result, we have to physically (or virtually) modify the executable at an assembly level to make changes.

At this point a few alarms and whistles may be going off for a few readers. What legitimate reason would we ever have to do so, modify an existing program for which no source is available? Consider the following hypothetical, but not too farfetched, scenario:

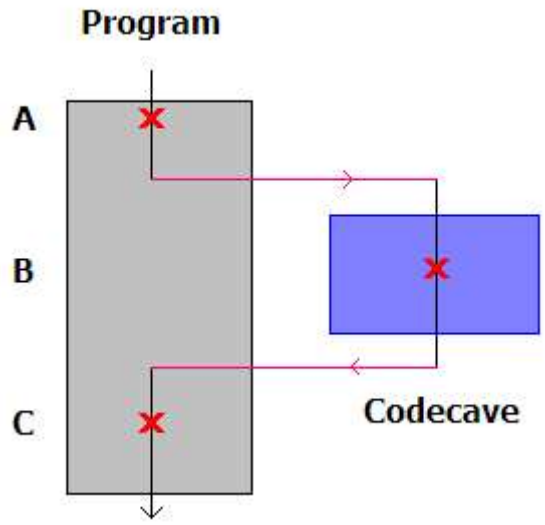
> A company that has been using the same software system they developed for the past 10 years. The software system they are using has served them well, but it is time to upgrade it to reflect a mandatory change in the output data format. The only problem is the original programmers are long gone and there are no hopes of getting the original source code to update the program. Now, this company has trained it's now veteran employees and grown the past 10 years using this specific software system, so a complete rewrite would be quite disastrous to the company. Retraining all their employees to a new system and having to reprogram things differently is not only time consuming but very costly. It would take about a year to do such and this is out of the time frame that the company has. The worst part of it all is that **you** are the programmer that was hired to solve this issue.

You could just throw up your hands and say it is not possible, but that would not do much to help your professional career. Instead, imagine if there was a way that you could keep using the same program, but you have an additional DLL that is used to dynamically update the output data from the company's program so it fits the new standard that is required. Best of all, it is a solution that can be implemented well before your deadline and requires minimal changes to be made to the company's existing procedures of using the program. Enter codecaves.

Now that we have a definition of codecaves and a purpose as to why we might ever need one, it is time for a visual presentation of this concept. Consider the following image as normal program flow:



In the image above, execution flows through the three points marked A, B, and C respectively in that order. The points A, B, and C are just any given areas of code; they could be one line of assembly code or multiple lines of assembly code. The point is (no pun intended), execution passes through them all at some point in time. If we were to codecave point B, we would end up with the following image:



In this image, execution first passes through A as usual. However, it is then directed into the codecave, which then executes in place of point B. After that execution is competed, the program execution is redirected back to after B and continues on through C. If you look closely at the picture, it may be a little vague as to **where** the codecave is in relation to the program. This is done intentionally just to show the concept first, therefore reference points of A, B, and C are used and nothing code related is shown. Hopefully, the concept of what a codecave actually is makes more sense now.

Codecave Attributes

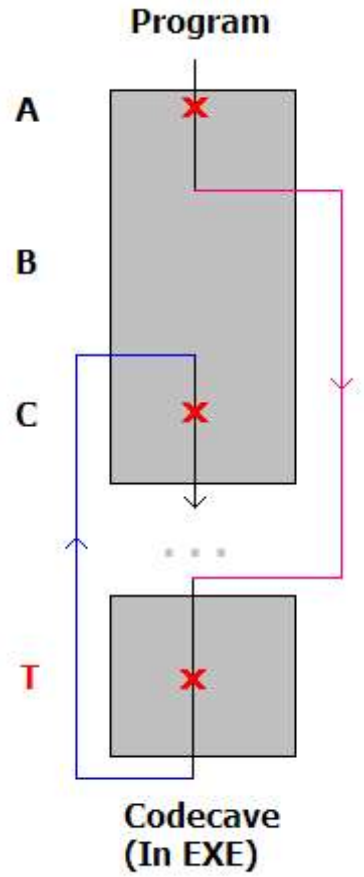
Now that we understand what a codecave is a little better, we can move into the different attributes of codecaves that you will need to be familiar with. I will discuss three main attributes that we need to be aware of and understand before we can use them. The first is the codecave location, or more simply where the codecave is implemented. The second is the codecave entry and exit points, which refers to how we get from the EXE into the codecave as well as from the codecave back into the EXE at the proper location. The final attribute is the codecave stack and register modification, which discusses the different logics we can use to make sure we do not modify the final stack and registers in the codecave.

Attribute 1: Codecave Location

The first attribute is the location of a codecave. This attribute of a codecave describes where the codecave is implemented. There are two parts to this attribute, a general location description and a specific location description.

Physical Location

Since codecaves must be in the **process space of the application**, there are two possibilities, in the EXE or in a loaded DLL. When a codecave is in the EXE, it is usually coded inline. This simply means the codecave is placed somewhere in an unused portion of the EXE that is empty or not used on a regular basics, such as exception handling code. Here is an example of a codecave that is contained in the program itself:



In this image, execution never leaves the program's module for the execution of the codecave. The codecave is placed in some area of the EXE marked "T" that is assumed to be suitable for a codecave. There are several advantages and disadvantages to this approach. The advantages are that it is very fast to implement, efficient, and easy to test and distribute. The disadvantages are that you must modify the EXE itself, it is not flexible, and you must code in assembly. Let us carefully take a look at each set of advantages and disadvantages now.

Advantages

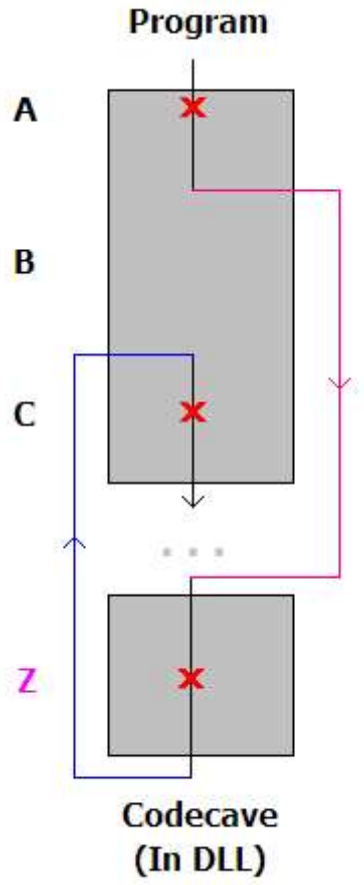
- **Fast to implement** - This advantage comes from the fact that there are disassemblers that are freely available which allow you to modify a program and save the changes immediately (such as OllyDbg). With this approach, you can locate a suitable location in the EXE for the codecave, make the appropriate changes to implement the codecave (not yet discussed in this article), and save the final EXE.
- **Efficient** - By placing the codecave in the EXE, this approach is efficient in the sense that you do not have to load anything additional into the program to implement your changes. As a result, this leads to the disadvantage of flexibility, but as long as you are using base API functions that the program imports and not doing any complicated logic, this point holds.
- **Easy to test and distribute** - Since you are modifying the EXE itself and saving it, you can fire it up in a debugger and set breakpoints before the codecave is executed to verify everything is working correctly. When everything is working well, you can just distribute the EXE and replace the original one. You could perhaps even write a file patcher that patches the original EXE into this final one that works to cut down on the distributed file size.

Disadvantages

- **EXE modification** - The first disadvantage comes from the fact that you **do** have to modify the EXE itself. If there is any physical file CRC check in place, then this approach will require those checks to be bypassed or faked. There is limited space in an EXE and there might not be a place that is large enough for you to place your desired codecave at. It is hard to figure out a "safe" place to overwrite, so placing new code over old might lead to future program instability.
- **Not flexible** - This approach is not flexible since any updates to the EXE that are made will require the codecave to be updated in a new EXE. It is also not flexible in respect to the next disadvantage.

- **Must code in assembly language** - Since you are modifying the EXE itself, you must implement all your code in assembly language. This may or may not be a disadvantage for you depending on what you need to do, but for the most part, this is a big problem since assembly language code takes up quite a bit of space.

Now that we know the basics for a codecave in the EXE, we can take a look at the second option for the location attribute, and that is implementing a codecave in a DLL. I will reemphasize the fact that codecaves must be in the **process space of the application**, so the visual image of a codecave in a DLL is the same as it is for an EXE. Nevertheless, here is an image of it just for completeness:



In this image, program execution will leave the program module for the execution of the codecave. The codecave is located inside some loaded DLL in some region marked as "Z". The advantages and disadvantages to placing a codecave in the DLL are the converse to those in the EXE. The advantages are that this method is very flexible, programmable in a higher level language, and dynamic in nature. The disadvantages are that it takes a bit longer to implement, adds overhead to the process, and is more difficult to test and distribute. Once again, we will take a look at each set of advantages and disadvantages.

Advantages

- **Flexible** - This approach is very flexible because any code that changes in the EXE can easily be updated in the codecave inside the DLL. If locations of the location of where the codecave belongs change, that too is easily updated by the DLL Loader that is used to get the DLL into the process. Refer to the disadvantages that result from this side effect as well.
- **Programmable in higher level languages** - Since the codecave is now in a DLL that we create, we are free to code our DLL into a higher level language such as C/C++/C++ CLI. This allows us to have more control and implement more complicated logic than is possible in the previous means of placing the codecave in the EXE itself.
- **Dynamic nature -** Since this method does not require the EXE to be physically changed, we can virtually enable and disable our codecaves if setup properly. This leads us to be able to create "one shot" codecaves that are executed only once and restore the bytes that once were written to lead into the codecave itself. Due to the other advantages, more complicated conditions of execution can be implemented as well to allow codecaves to be executed only under certain conditions.

Disadvantages

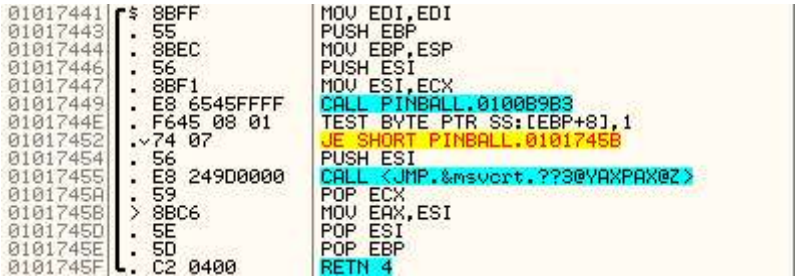
- **Longer implementation time** - A big issue in this method is that you first have to develop your codecave in a DLL, then load the DLL into the program, and finally make the EXE redirect into your codecaves. If you are new to this topic, getting your DLL "just right" so the codecave works as expected takes a bit of time, but once you are used to it, this will not be as big of a disadvantage.
- **Additional overhead** - This is an obvious disadvantage but still important to mention. Since you are loading an additional DLL into the program and executing more code than there was before, you might degrade performance if you do not correctly implement your logic. For the most part, you will have a negligible impact on the program's performance, but only if you do not write inefficient code that is executed inside the codecave.
- **More difficult to test and distribute** - This disadvantage correlates to the flexible advantage above. Since the codecave is in a DLL, you must write a DLL loader to get the DLL into the program. This "DLL injection" makes it so you cannot easily launch the target process with a debugger. Instead, you must attach it to a running process. This leads to some tricky cases if you must debug the codecave when the EXE is just starting.

Now we know the two main options we have for where to implement our codecaves. We can place them in the EXE with quick and easy convenience to test out relatively simple logic. If we need a bit more of flexibility and power we can resort to implementing the codecave in a DLL.

Logical Location

The second part of this attribute that has to be discussed is the location of the codecave itself in relation to an assembly listing. The guideline that we must follow is that we will almost always need at least **5 bytes** of space to setup a codecave using one of the methods discussed in the next section to get into the codecave. Wherever we place the codecave, we must be able to easily restore the bytes that were overwritten inside our codecave. This means instructions that involve PUSH, MOV, CMP, or unconditional long JMPs are the most ideal locations to place a codecave if it is 5 bytes or more. We want to find a location that we will have to do the least additional work to restore. This paragraph is a bit hard to digest, so here are some examples and explanations of what is meant.

Consider the following code listings of an arbitrary function from the Pinball game:



Let us say that we need to set a codecave somewhere in that function to get access to some arbitrary data elsewhere in the program. The only condition we have is that the codecave must fall in this function; we can get access to what we need once the codecave is setup. Where is the "best choice" location for our codecave?

Since we only need at least 5 bytes of space to place a codecave, we could put it in the first 3 instructions, which take up exactly 5 bytes. However, that means we would have to implement that code in our codecave. "That code", being:

Hide Copy Code

```
01017441  /$ 8BFF           MOV EDI,EDI
01017443  |. 55              PUSH EBP
01017444  |. 8BEC            MOV EBP,ESP
```

This solution would work out well for us because there is nothing version specific about this assembly code. Assembly code at the beginning of the function is usually generated by the compiler (referred to as prolog code) and remains constants between all versions. Since this code is also always guaranteed to be executed, it would make for a great choice to place a codecave. However, let us consider the other possibilities. If we look at the following line:

Hide Copy Code

```
01017449  |. E8 6545FFFF    CALL PINBALL.0100B9B3
```

This instruction takes up exactly 5 bytes as well, so would this make a good candidate? The answer is, "it depends". If there was no other suitable location for a codecave and we had to choose this place, it would be fine. However, there are some drawbacks to using this location. If the program should change during an update, chances are the `CALL` address would be different. This would result in the necessity to update the codecave address as well as the code that is called inside the codecave. In short, we have more work to do in the long run.

If we look at the next two lines:

Hide Copy Code

```
0101744E  |. F645 08 01     TEST BYTE PTR SS:[EBP+8],1
01017452  |. 74 07          JE SHORT PINBALL.0101745B
```

We can see the total combined takes up 6 bytes, which is more than 5, but a conditional jump is part of the logic. We want to avoid placing a codecave in code that involves conditional jumps at all costs. They add a lot of work to properly reprogram in our codecave. How about the call made after the conditional jump?

Hide Copy Code

```
01017455  |. E8 249D0000    CALL <JMP.&msvcrt.??3@YAXPAX@Z>
```

For similar reasons to the CALL above, this line carries the same drawbacks that on a program update, the address might change thus creating us more work, but the real no-no in this line is that it is not always going to be executed! This code could be the error condition of the comparison made above, so our codecave would never be executed. We have to pay particular attention to conditions such as these when looking for a location to codecave.

Finally, at the end of the function we have a set of 5 bytes made up by the instructions:

Hide Copy Code

```
0101745A  |. 59             POP ECX
0101745B  |> 8BC6           MOV EAX,ESI
0101745D  |. 5E             POP ESI
0101745E  |. 5D             POP EBP
```

Would this be a suitable codecave location? **Absolutely not!** This example right here is the minefield of codecaves that leads you down the path of crashing your program. If you look above at the conditional jump, it will jump to the line `MOV EAX, ESI`. This means that when the conditional jump is taken, the program will execute an invalid byte sequence since our codecave started one byte before. This scenario is very important to be on the lookout for.

After looking at most of the possibilities of where to place the codecave, the best one would be at the beginning of the function. In this example, we had a lax requirement of only needing to have a codecave in that location. This is a best case example that rarely happens. Most of the time, we have a particular place the codecave has to be at, so we must carefully read the code to make sure we will not crash the program if we place a codecave there.

The main bit of information to take from this example is that you must carefully read the assembly listing code in full context before you select a location for placing a codecave. With this attribute fully covered now, we can move on to the next attribute, which involves the ways we can actually get into our codecave.

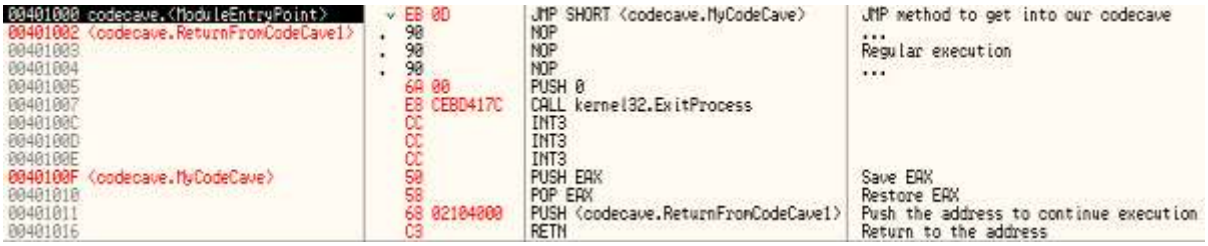
Attribute 2: Codecave Entry and Exit

The second attribute of codecaves that must be covered is the actual entry and exit points of the codecave. We must be able to get the EXE to execute our codecave or we will just have added useless code. For this attribute, there are two ways we can go about this, both of which involve indirectly modifying the Instruction Pointer in the program. The first method is to use a `JMP` instruction to directly jump into the codecave. The second method that is possible is to use a `CALL` instruction to call the codecave.

JMP Method

The `JMP` method is the easier method of the two to use since we can directly `JMP` to the codecave and then `JMP` back through a `PUSH` and `RETN` to where we belong. The advantage of this method is that the stack is preserved when we make the `JMP`, so as

soon as we are in the codecave, it is as if we had not made the JMP at all. This is important since we must execute the code that was overwritten in the codecave location in the first place. If we are not careful with the stack, we might crash the program if things get out of order. The disadvantage of this approach is that the return address must be hardcoded in the codecave. If the EXE is not going to change, there is nothing wrong with this approach, however, if it does change often, it will get annoying to update the return address each time. It is easy to forget to update the return address and end up crashing the program since you are returning to a bogus location. There will be more on this disadvantage later on. Here is an example of this method in action:



If you are not used to OllyDbg, it may take a bit of reading the image to understand fully, but there is no rush. Read it over carefully and continue once you get what is going on. The left column is the addresses. Labels have been added to help with the important address identification. The middle two columns are the HEX and ASM code listings respectively. The forth column is additional comments. This example is only to show a quick means of how this method works. As a side note, the first `JMP` usually will not be a `SHORT JMP`, but a `LONG JMP`, since the codecave will not be so close to the location that is codecaved.

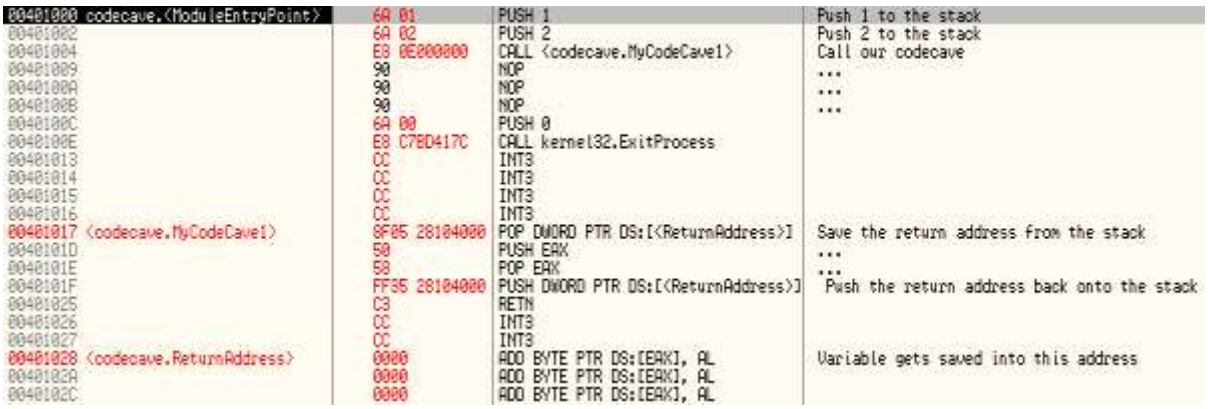
As mentioned above in the disadvantage, it is your job to maintain the codecave to return back to the correct location to continue execution. In the above image, that is at address `0x401002`. If code is inserted before that location, then we would have to update the codecave to `PUSH` the new return address before the codecave returns. Otherwise, we will be returning back into an invalid code sequence. In accordance with the first attribute, if you may have noticed, this example was done using the codecave in the EXE itself. It is a bit harder to completely present the alternative method visually, so that is why I chose this way.

That is all there is to the `JMP` method. It is quite easy to implement and use, but requires a bit more maintenance for when the EXE changes.

CALL Method

The `CALL` method is a little more complex than the `JMP` method because we must save the return address from the top of the stack before we execute our codecave and then push it back onto the stack before we return. The advantage to this method is that we do not have to maintain the return address ourselves since it is pushed onto the stack to begin with. If we do not, all of our codecave operations have to operate in respect to a stack pointer that is 4 bytes off from what is expected. That is a mess to work with, so it is better if we take care of it ourselves.

Now, that last paragraph will either make total sense to you or will leave you scratching your head. Let me assume the latter and present some pictures to illustrate what I just said. Let us assume this is our original code presented in this image:



If we trace up to the `CALL`, our stack will look like this:

```
0006FFBC  00000002
0006FFC0  00000001
0006FFC4  7C816FD7  RETURN to kernel32.7C816FD7
0006FFC8  7C910738  ntdll.7C910738
0006FFCC  FFFFFFFF
0006FFD0  7FFDF000
0006FFD4  80543FFD
0006FFD8  0006FFC8
0006FFDC  887D4B70
0006FFE0  FFFFFFFF  End of SEH chain
0006FFE4  7C839AA8  SE handler
0006FFE8  7C816FE0  kernel32.7C816FE0
0006FFEC  00000000
0006FFF0  00000000
0006FFF4  00000000
0006FFF8  00401000  codecave.<ModuleEntryPoint>
0006FFFC  00000000
```

If everything works out well, the call stack will look the same when we make the `CALL` as it would in the `JMP` method (this will not be the case):

```
0006FFB8  00401009  RETURN to codecave.<ModuleEntryPoint>+9 from codecave.00401017
0006FFBC  00000002
0006FFC0  00000001
0006FFC4  7C816FD7  RETURN to kernel32.7C816FD7
0006FFC8  7C910738  ntdll.7C910738
0006FFCC  FFFFFFFF
0006FFD0  7FFDF000
0006FFD4  80543FFD
0006FFD8  0006FFC8
0006FFDC  887D4B70
0006FFE0  FFFFFFFF  End of SEH chain
0006FFE4  7C839AA8  SE handler
0006FFE8  7C816FE0  kernel32.7C816FE0
0006FFEC  00000000
0006FFF0  00000000
0006FFF4  00000000
0006FFF8  00401000  codecave.<ModuleEntryPoint>
0006FFFC  00000000
```

If we look carefully, the top of the stack now has the return address from the `CALL` we made. As a result, we must pop off this value. However, we cannot simply discard it; we must **save** it so we can return to this address when we are done with the codecave. That is why we have the line of code that POPs off the top of the stack into our variable (the instruction at address `0x00401017`). Once that line executes, the top value of the stack is stored into our variable and the stack will look like it should, and that is with the top being `0x6FFBC` point to rather than `0x6FFB8`, which points to the return address for the `CALL`.

Once we have taken care of this slight inconvenience, we are free to continue on just as if we were in the `JMP` method.
This `CALL` method also offers a bit of flexibility since the return address is already saved, we can actually modify that with various offsets so we can return wherever we want. Assuming the EXE changes and the function we are codecaving inside does not change, the offsets are still valid! That is not to say we could not do the same in the `JMP` method, but it would be a little more work for that extra complicated logic.

With these two methods covered, we know the theory of how to get into our codecave from the EXE and back to the EXE from the codecave. It should be noted that while the above examples were done with the codecave in the EXE, it would look the same as if it were in the DLL, except the addresses would be different. We have one more attribute left to cover now and that deals with the theory of working with the registers and stack inside the codecave.

Attribute 3: Codecave Stack/Register Modification

The last attribute of codecaves that we have to be aware of is the stack and register modification that takes place inside the codecave. This attribute is an extremely important aspect that must be carefully observed when designing your codecaves, or disastrous results may occur. Take for example the next snippet of code:

```
00401000 codecave.<ModuleEntryPoint>   6A 00          PUSH 0
00401002                               B8 DACD817C    MOV EAX, kernel32.ExitProcess
00401007                               EB 0F          JMP SHORT <codecave.MyCodeCave>
00401009 <codecave.ReturnFromCodeCave> 90             NOP
0040100A                               90             NOP
0040100B                               90             NOP
0040100C                               FFD0           CALL NEAR EAX
0040100E                               90             NOP
0040100F                               90             NOP
00401010                               90             NOP
00401011                               90             NOP
00401012                               90             NOP
00401013                               90             NOP
00401014                               90             NOP
00401015                               90             NOP
00401016                               90             NOP
00401017                               90             NOP
00401018 <codecave.MyCodeCave>         B8 00000000    MOV EAX, 0
0040101D                               68 09104000    PUSH <codecave.ReturnFromCodeCave>
00401022                               C3             RETN
```

Big oops! In our codecave, we modified the EAX register, so when the codecave returns, we will execute a CALL 0 instruction, which will result in an exception and a crash in the application. Now it is obvious why we must pay particular attention to the stack and registers. This example is pretty trivial, but when you are working with more complex code, you may inadvertently modify the stack or a register and crash out your program. There are a few things we can do to help preserve the stack and registers as we go along in our codecave.

PUSH/POP

We can use this instruction set pair to preserve one register at a time. For example, if we need to use only the ECX register, we can do the following:

Hide Copy Code

```
push ecx
  ...
  ; Use ecx
  ...
pop ecx
```

The advantage of this instruction set is that we can only save and restore what we need if we are sure that we only need to modify a few registers. If we need to access the stack contents, we know we only have to modify it by 4, so it is an acceptable calculation to make.

PUSHAD/POPAD

We can use this instruction set pair to preserve all general purpose registers at the same time. We will want to use this if we were calling a function from inside our codecave that might modify one or more registers.

Hide Copy Code

```
pushad
  ...
  call MyFunction      ; Modifies quite a few registers
  ...
popad
```

It is very important to remember that the PUSHAD instruction modified the stack with 8 32bit values, so if you have to access or use the stack, you will want to do it **before** you actually use a PUSHAD instruction. Otherwise, you will have to modify the stack accessing by 0x20 each time. Here is an example of how the stack looks before a PUSHAD is executed and then afterwards. Note how much the stack has changed:

In the original stack, the topmost value of -1 is accessed as ESP + 8. After the PUSHAD, we must use ESP + 0x28. This is what is meant by recalculating the stack address in the codecaves if we modify the stack. Also note that $ is just ESP.

PUSHFD/POPFD

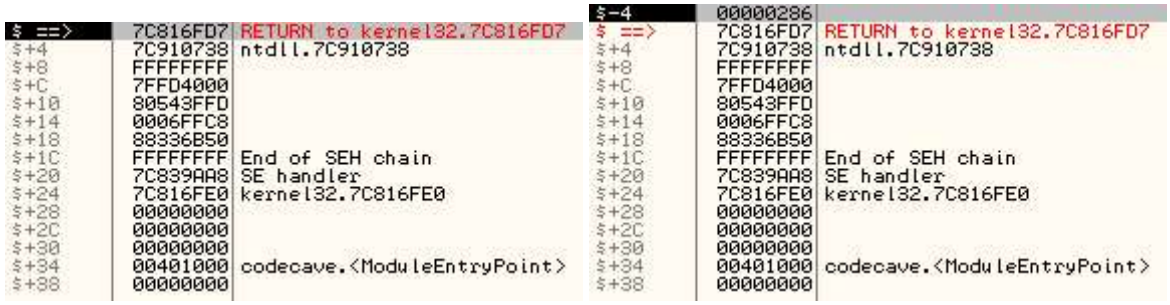
This instruction set pair is similar to the PUSHAD and POPAD pair, but it saves the flags to the stack. This instruction set only saves one 32bit value onto the stack, so you work with updating stack access calculations just like you would with a PUSH, you just add

4.

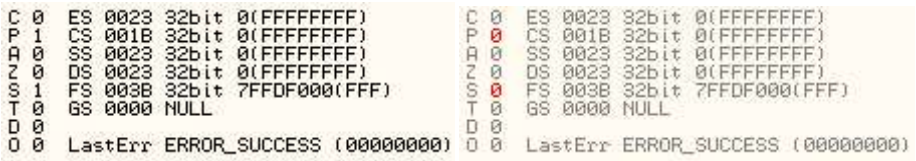
Hide Copy Code

```
pushfd
  ...
  test eax, eax
  ...
popfd
```

Here's an example of the stack before a PUSHFD is executed and then afterwards:



As you can see, the stack has only been modified by 4 bytes. Note that I did not change the address reference in the second picture above, so that is why the topmost address is ESP - 4 rather than ESP as the second picture is in the PUSHAD example. Here's a look at the flags themselves before the TEST EAX, EAX is executed in my example and after:



After the POPFD is executed, the flags are restored to the original states seen in the first image above.

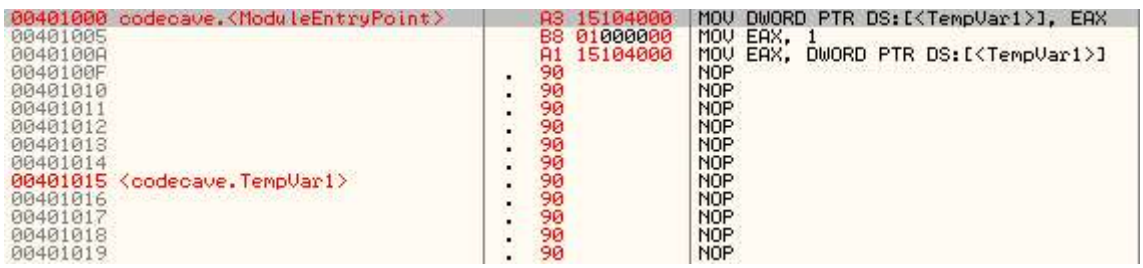
Temporary Storage

The last technique we can use to help preserve the stack and registers is to use temporary variable storage. Rather than store an original register on the stack, we can save it to a memory location, use the register, and then restore it back ourselves.

Hide Copy Code

```
mov [VariableAddr], EAX
  ...
  ; Use EAX
  ...
mov EAX, [VariableAddr]
```

Here is a visual example:



In this code, we first move the EAX register into the memory location, use EAX, and restore it back from the saved memory location. By doing this, we do not modify the stack at all, so we do not have to change any stack calculations! It is a bit more work and we must ensure that we restore the right variables back into registers, but aside from that, it is an interesting approach to take.

There is one caveat that I must bring to your attention here. This section was written in regards to beginning codecave writing. In this case, it is important for a beginner not to modify the stack or registers. However, as more experience is gained, you will find

at times that you do want to modify the stack or registers. If you want to intentionally modify the stack or registers, this is fine. Just make sure you know what you are doing before you do it.

Now that the three main attributes of codecaves have been discussed, we have a more complete knowledge and understanding of codecaves combined with the earlier knowledge of the basic theory. What is left now is to take a look at the practical application of codecaves and get a good feel of how to develop and use them in a real world example.

Application

With the theory of codecaves covered, we can now put that knowledge to use. Before we continue though, there are some tools we need to obtain. Once we have our set of tools, we have to then get the program that we will be working on. For this article I choose to use the Space Cadet Pinball game that comes with WinXP. Reading over the EULA of Windows components, I don't think I can redistribute the exact version I have. I will provide a little tip to help those that have different versions when we get to working with the actual program. Once we collect the tools discussed in the next section, we can move on to making a codecave! Here is an image of the version of the Pinball game that I will be using:

Tools

The first tool we need is a memory scanner. For this tool, I will recommend TSearch. The purpose of this tool is to scan the memory of a process to find the address of data. There is a link to the program on the Wikipedia page, but be sure that you scan it carefully, as with anything you download on the internet. The next tool we need is a disassembler and debugger. For this, I will recommend that you use OllyDbg since it is free, powerful, and easy to use. The final tool we will need is a C++ compiler. For this I will recommend that you use Visual C++ if you have it. If you do not, you can get the Express Edition. Please note that another free alternative such as Dev-Cpp will work, but the source code is not compatible with it since it does not use Intel style inline assembly. To recap, we will need three tools: a memory scanner, a disassembler, and a C++ compiler. If you have never used any of these tools before, this section of the article might take a bit more time for you to understand. You might want to reference some additional tutorials on these tools if the images and text provided here are not enough for you. Just remember to be patient and reread the places that you get stuck at. This stuff is by no means easy. Here is additional overview information (taken from Wikipedia) on the tools we will be using. I did not include links to tutorials and additional resources due the nature of the sites they are hosted on.

TSearch

> "TSearch (similar to ArtMoney and the open source Cheat Engine) is a memory scanner/debugger utility developed by Corsica Productions. TSearch's primary function is to scan open processes for byte addresses; restricting searches to either 'Exact Value', 'Range' or 'Unknown Values'. The searches can be refined by using a 'Search Next' (or sieve) option: this re-searches already found results to display results suiting a further refined criterion. TSearch also features a hex editor and 'auto hack' option, and is commonly used within the game hacking community to develop 3rd party game 'trainers' and 'hacks'."

OllyDbg

> "OllyDbg is a debugger that emphasizes binary code analysis, which is useful when source code is not available. It traces registers, recognizes procedures, API calls, switches, tables, constants and strings, as well as locates routines from object files and libraries. According to the program's help file, version 1.10 is the final 1.x release. Version 2.0 is in development and is being written from the ground up. The software is free of cost, but the shareware license requires users to register with the author."

Visual C++

> "Microsoft Visual C++ (also known as MSVC) is an Integrated development environment (IDE) product engineered by Microsoft for the C, C++, and C++/CLI programming languages. It has tools for developing and debugging C++ code, especially that is written for the Microsoft Windows API, the DirectX API, and the Microsoft .NET Framework."

Work Process

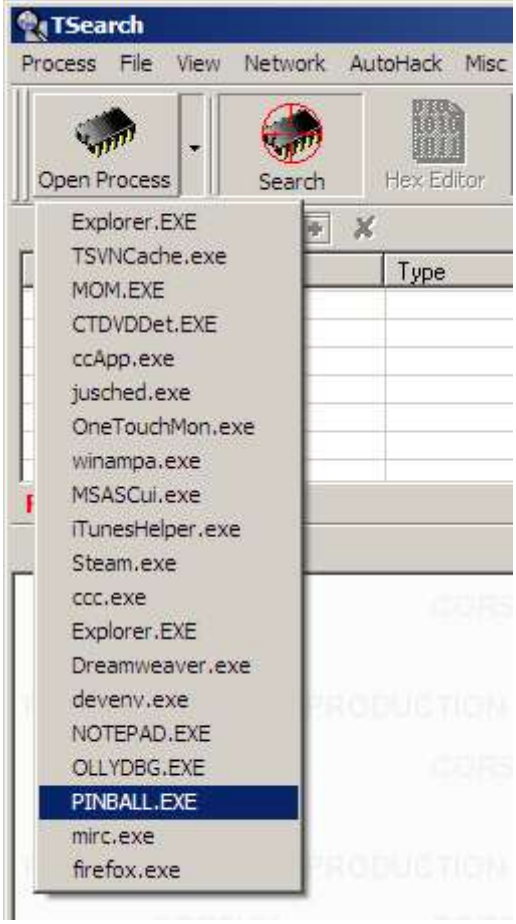
Now that we have the tools we will be using and our target program, we can get started with applying the theory of codecaves to get something done. What follows are the three main steps we must follow. The first step is to find a purpose for the codecave we wish to create. Usually, you would already know why you need a codecave, but for the sake of the article, we will work through this step from the beginning. The next step is writing the actual codecave. For this article, we will write a codecave that is located in a DLL and uses the `CALL` method. The last step is putting it all together to get our DLL into the process and watch our codecave in action.

Step 1: Finding a Purpose

Let us start up the Pinball game and take a look at what is there for us to mess with. Here is an image of the main game screen for reference:

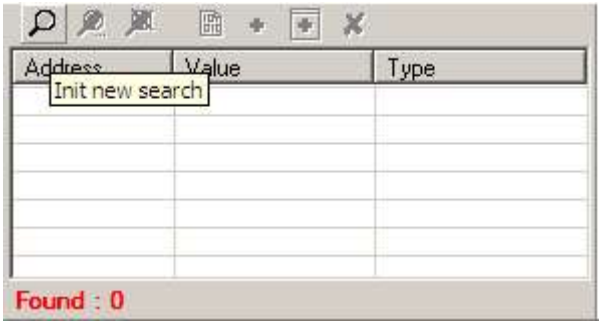
If we consider what is displayed and what changes the most, the only thing we really have to work with is the current score. What if we wanted to make a program that displayed the current score for us outside of the program? We could then do some sort of score logging or other statistical analysis if we wished. Either way, we are trying to do something that cannot currently be done since only the game knows the current score. We could just read it from memory, but in some cases, such things might not be possible, so we will assume our only option is to make a codecave to extract out the data.

As a quick recap for step 1, we will define our purpose to be to create a codecave in the Pinball game so we have access to the score. Now that we know what we want to do, we have to start our reverse engineering procedures to figure out how to actually find a suitable location in the program to extract the score from. To do this, we will use TSearch. Go ahead now and start up TSearch. In the menu bar at the top, select the Open Process button and choose our program, "*Pinball.exe*".

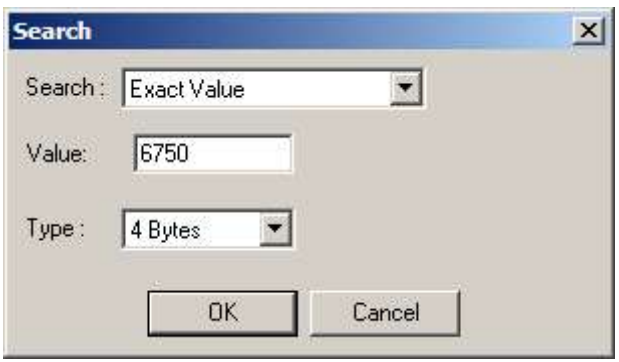


Once you have done that, the title of the window will change to "TSearch - *PINBALL.EXE*". This lets us know that the process has been loaded. With TSearch loaded, our first order of business is to find the address where our score is being held at. To do this, play the Pinball game a little to get a non-zero score and press F3 to pause the game. I stopped when my score was 6750. Once you have your score handy, switch back to TSearch, we will search for the memory address of the score.

Look at the top leftmost area that contains a grid control. We will use the first button to search for the score. Click on the first icon, the magnifying glass to start a new search.

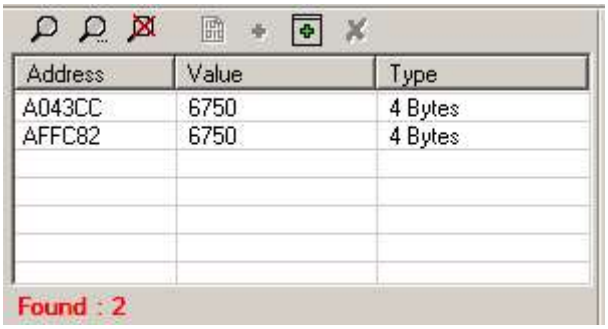


A new dialog will open that will allow us to select some data type options, search options, as well as actual values. Since the score is displayed as what looks like an integer, we will use the default search parameters to look for our score. In other programs or games, you might have to play around with the settings to find the right type for what you are looking for. Go ahead and enter your score and press OK.



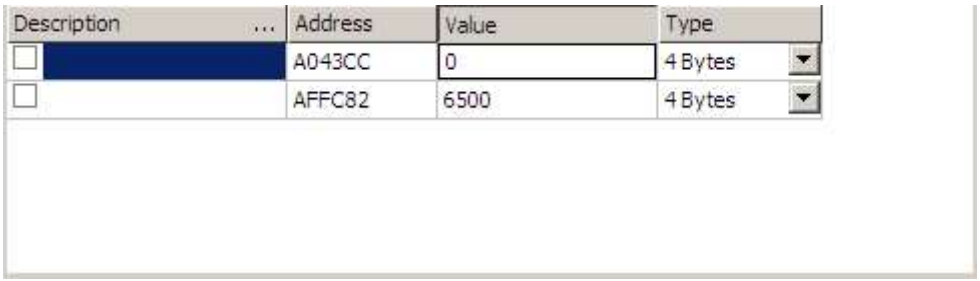
If everything goes well, TSearch will scan the program's memory and tell us the addresses of data that match our score. In my case, it was exactly 2. Depending on the value, you might get more. If you do get more, I would change your score by playing a little more and searching again so you get exactly 2 results. Here is an image of the results dialog that pops up after a search:

After we hit OK and look back at the grid control that we used to search from, we can see two addresses being listed that contain our score:



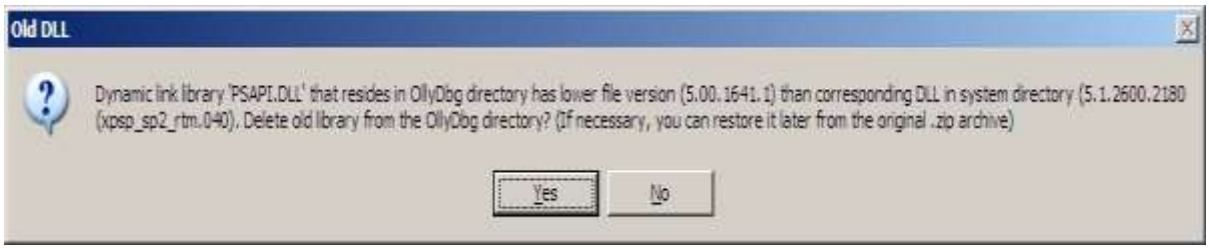
Now, why do we have two addresses listed? Considering that we have a variable to store the score, there is probably a second variable that holds the score to display it to the main GUI. As a result, there may be two or more values that contain the data we are looking for in other programs. This is something that is important to keep in mind. We know that one is the real score and one is a dummy score, how do we tell them apart? There is no easy answer other than trial and error. Referencing the image above, click on the button with the "boxed, green plus sign" that reads, "Add all line found to the table". After we click that button, the two entries are copied into the panel dialog on the right. We will now figure out which one is the real score.

We will now work from the right side of the screen where the two addresses were just added to. Click inside the column that reads Value of the first entry and change it to 0.

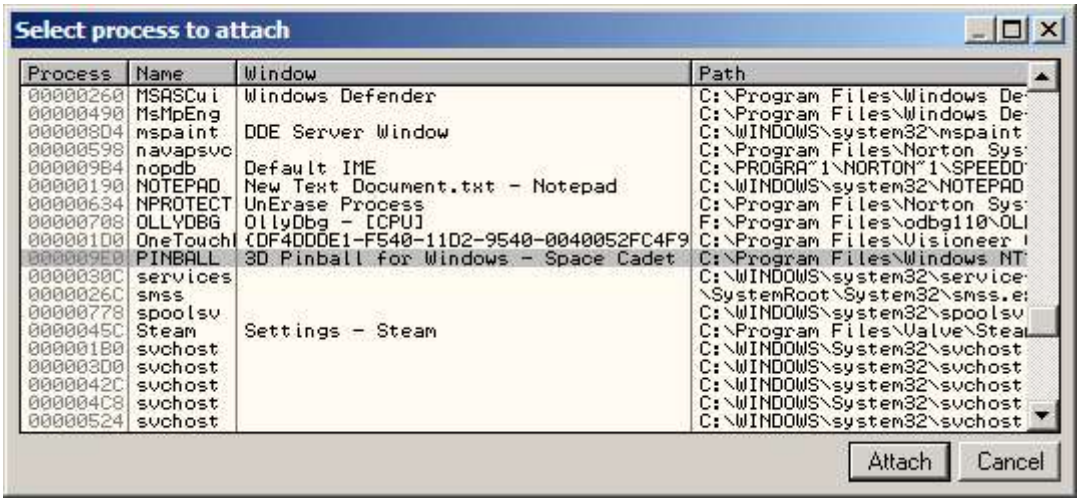


Switch to the game and play it until you get a few more points. If the memory layout is similar to mine, then you will see that your displayed score continues to go up and the memory location we just set to 0 is reset back to the current score. This means that address is **not** our real score. To verify this, hit F3 to pause the game again and go back to TSearch. In the second address, change its value to 0 and repeat the same procedure. Once we get a few more points in game, we will see our score is now reset, and we are playing from the beginning again. This is a clear indicator that we have found the **correct** score address.

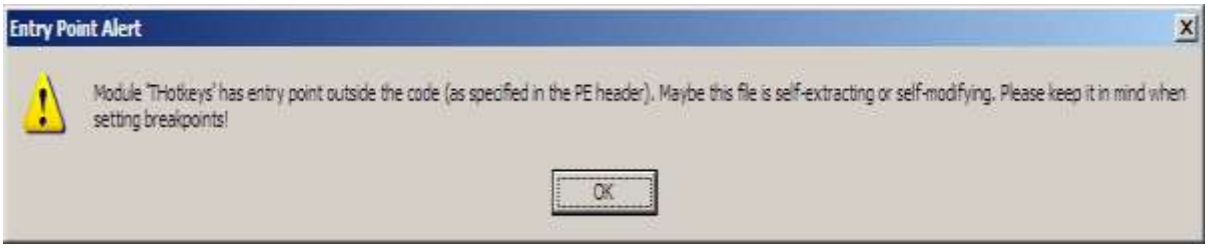
Now that we have the address of our score, we must find where in the EXE the game accesses or modifies it so we can setup a codecave at that location. To do this, we will attach OllyDbg to the process. Take this time to start up OllyDbg. If this is your first time launching the program, you will need to click "*Yes*" at the first message box if you get one:



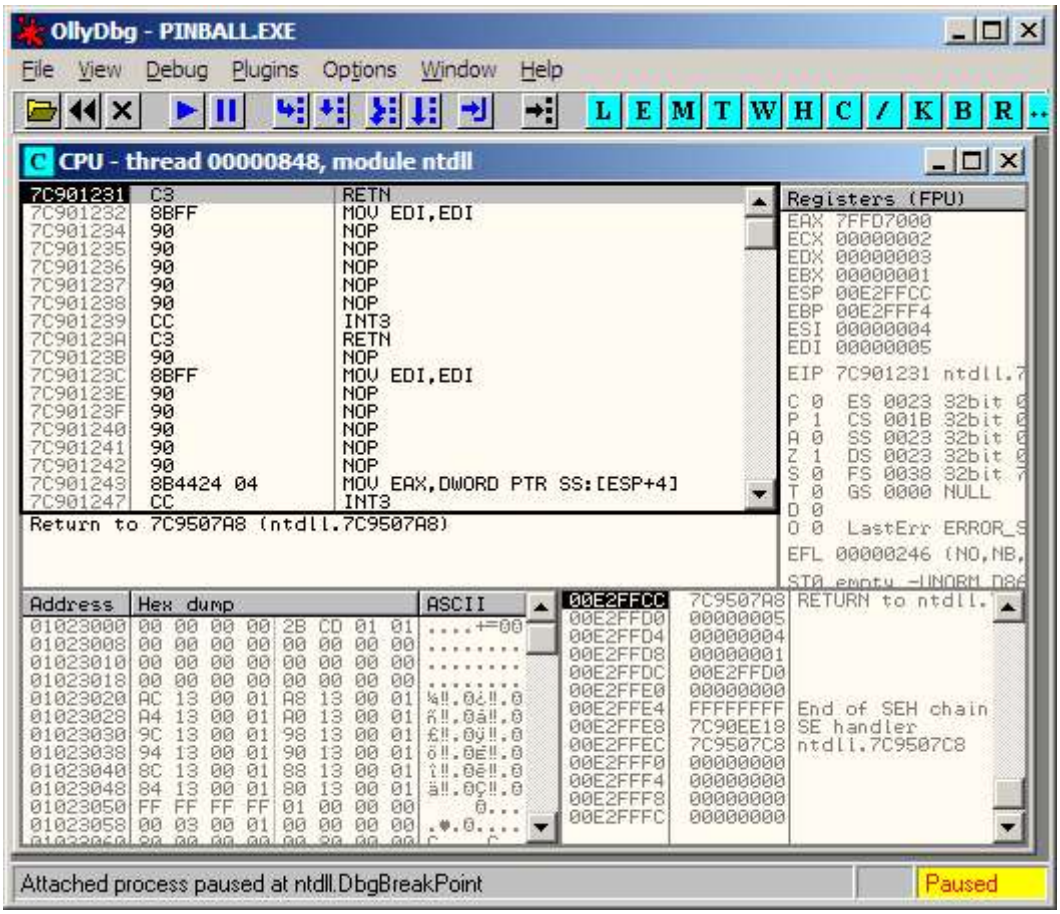
With OllyDbg open, click on "File" and then "Attach". Select the Pinball process from the list and click the Attach button. You can click on the Name heading to sort the list by name to make finding the process a lot easier.



As soon as you attach OllyDbg to the process, you will be greeted with another message box. Simply press OK.



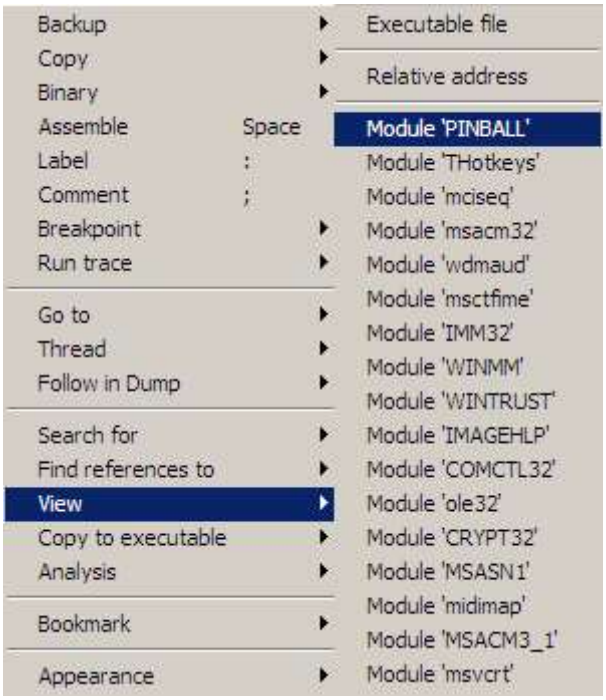
At this point, OllyDbg is attached to our program and we should see something like this:



There are a few important things to be aware of at this point. Right now our program is in the suspended state. We know this because at the bottom right corner in yellow, it says "Paused". The module that is currently being displayed is not our program, but the *ntdll.dll* file. This is foretold by the window caption, which says "module ntdl". We will first need to unpause our program

by pressing the F9 key or the blue 'Play' button (sideways triangle pointing to the right) at the top of the menu bar. You will know that you did it correctly if the bottom right corner changes from "Paused" to "Running".

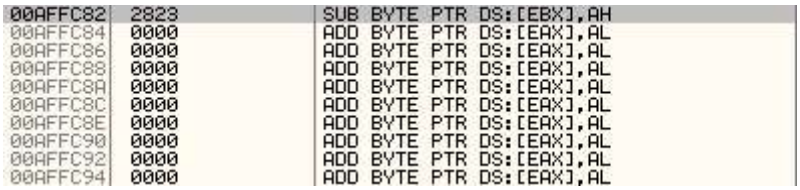
With the program running again, we need to actually look at the Pinball module rather than the *ntdll.dll* module. Right click on the assembly listing pane and choose "View" and then "Module 'Pinball'".



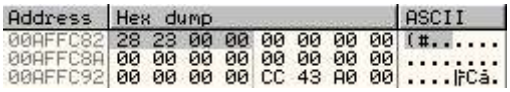
You must do this a few times until the main window says "CPU - main thread, module PINBALL". If you do it once and it still says "module ntdll", you will have to do it again! Once you are in the main PINBALL module, press "Ctrl + A" to do an analysis of the code. This will make things a little cleaner to read. Alternatively, right click in the assembly listing and choose "Analysis" and then "Analyze Code". You might also want to change the appearance of the code so it is not all black and white. Right click again in the assembly listing window and choose "Appearance", "HighLighting", and then "Jumps'n'Calls".

At this point, we have OllyDbg attached to the Pinball process and we know the address of our score from TSearch. Copy the real score's address from TSearch into the clipboard. Switch to OllyDbg and press "Ctrl + G" and paste in the address. Prefix the address with "0x" so OllyDbg knows it is an address since it starts with a letter.

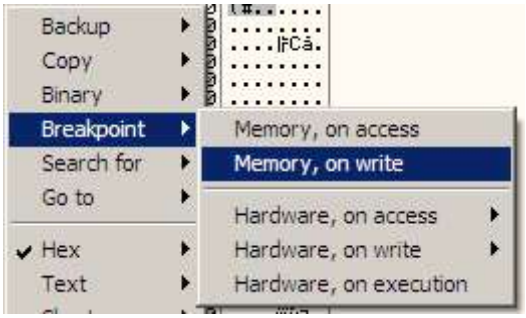
After you press OK, you will be taken to the address in the main disassembly dump window. You should see something that looks like this:



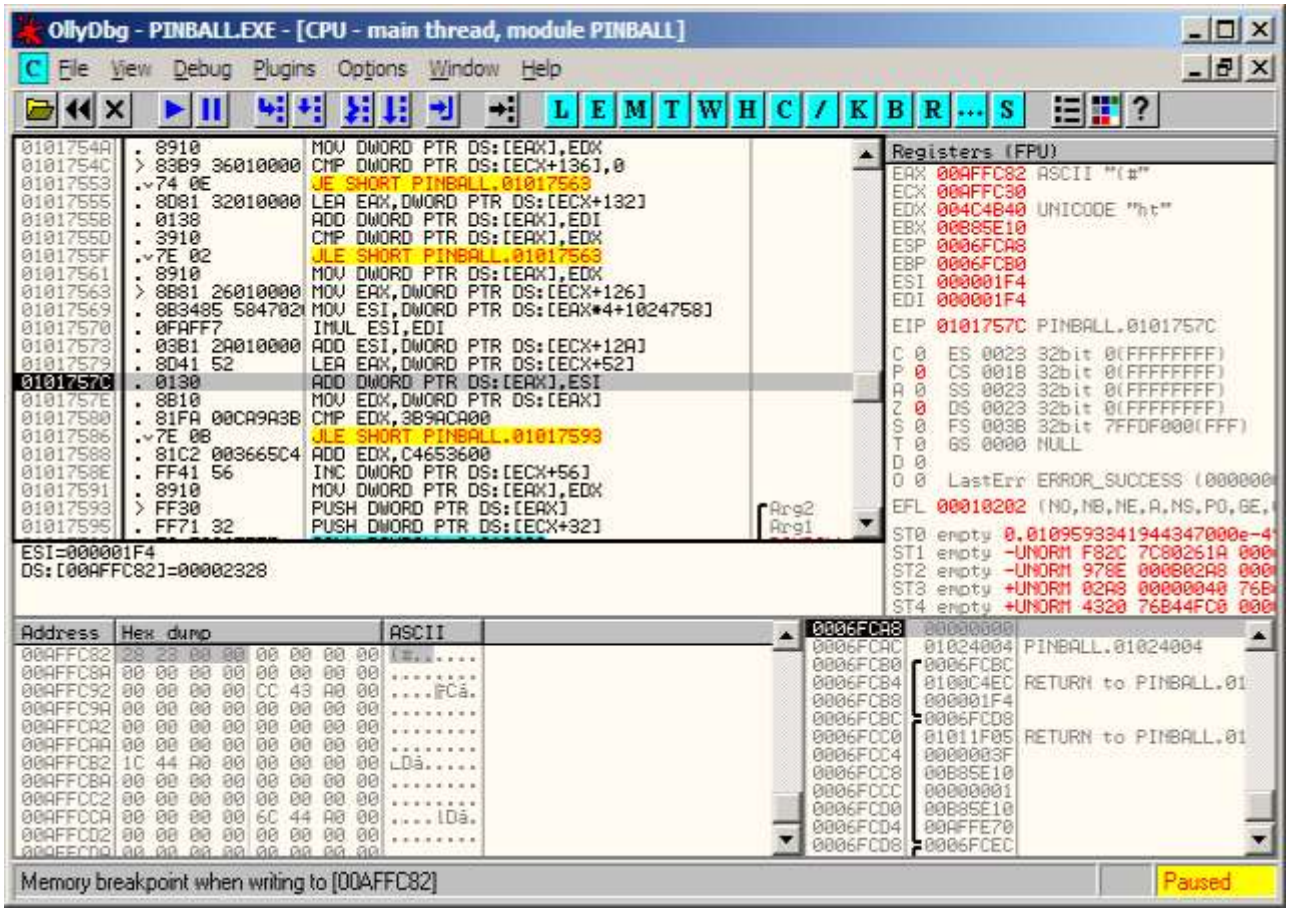
What you are seeing now is the disassembly of the memory contents at that location. However, this is not the view we want to look at this data. Make sure the main line is highlighted and right click. Choose "Follow in Dump" and then "Selection". In the bottom portion of the screen, you will see the data in a HEX view. Highlight the first 4 bytes, this is our score variable:



The next thing we have to do is set a memory breakpoint on this location so OllyDbg will pause the program when the game writes to this memory location to update our score. By doing this, we know of one location we can place a codecave at to get the current score. Right click on the 4 highlighted bytes and choose "Breakpoint" and then "Memory, on write":



With the memory breakpoint in place, we can switch back to the game and unpause it. As soon as you get points, the game should pause and the debugger should become active. Your debugger should look something like this when it is focused from the breakpoint:



Note in the bottom right corner the process is "Paused" again. In the status bar, the text reads "Memory breakpoint when writing to [00AFFC82]". This lets us know that the debugger has caught the process writing to our score's memory location. Let us take a look at the assembly instructions that pertain to the score being modified.

Hide Copy Code

```
01017579  |. 8D41 52          LEA EAX,DWORD PTR DS:[ECX+52]
0101757C  |. 0130             ADD DWORD PTR DS:[EAX],ESI
0101757E  |. 8B10             MOV EDX,DWORD PTR DS:[EAX]
01017580  |. 81FA 00CA9A3B    CMP EDX,3B9ACA00
```

Here is a rough translation of what is going on. By looking at the above, we know the score's address is loaded into `EAX` from `DS:[ECX + 0x52]`. Then, the `ESI` register is added to the score's value. Finally, the score variable is loaded into `EDX` and then a comparison is made to the static constant `0x3B9ACA00`, or decimal 1,000,000,000. Having seen at least one different version of the code above in the different Pinball versions, I will choose to place the codecave on the instruction `CMP EDX, 3B9ACA00`. Since that instruction is 7 bytes, it meets the requirements of a codecave needing at least 5 bytes. Furthermore, reprogramming

that logic in a codecave is simple; we just execute that line at the end of the codecave before we return to the EXE. Placing the codecave is really convenient at that location since we know the score is stored in `EDX`. We can simply move the register to our own variable for use without any additional work.

If your listing is a bit different than above, it is ok. As long as you see something that resembles the above and have some `CMP` instruction that compares a register against `0x3B9ACA00`, you are fine. If you did not break at a location such as this, you might have chosen the wrong address as the score variable. This is ok too since you can hit CTRL + F and do a search for `CMP EDX, 3B9ACA00`. You should land in the right area now. We should remember the address that the `CMP` instruction is on since we will be using that in the next step.

This is a good place to stop for a moment and recap what we have done so far. We first started out by looking at our Pinball game and found a task to do. We decided we wanted to be able to have access to the score outside of the game. Once we established that goal, we then moved into finding the address of that data in the process using TSearch. Once we correctly identified the location, we used OllyDbg to find where the process modifies the address, which in turn gives us a location to codecave to extract out the current score. With all of that done, we have completed step 1. Now, we can move on to the next step, which is to write the codecave itself.

Step 2: Writing the Codecave

At this point in the article, a slight problem arises. It will not be a problem to implement our codecave, but it will be a problem to actually write it to the process. The concept of dynamically changing a loaded program's code is another article in itself. Because of this limitation, I will provide the basic functions we will use to accomplish our goals for this article. It will be up to you to research more into them to fully understand what they do and why things are the way they are (something that takes a bit of time and practice). The code is well commented though so only a few concepts need to be explained.

We will generally have at least two functions for each codecave we make. The first function will be the codecave itself. The remaining functions are support functions that are called from the first function to handle additional logic that cannot go in the first function. The codecave function itself has to be a special type of function, a **naked function**. Here is an excerpt taken from MSDN:

> Functions declared with the **naked** attribute are emitted without prolog or epilog code, enabling you to write your own custom prolog/epilog sequences using the inline assembler. Naked functions are provided as an advanced feature. They enable you to declare a function that is being called from a context other than C/C++, and thus make different assumptions about where parameters are, or which registers are preserved. Examples include routines such as interrupt handlers. This feature is particularly useful for writers of virtual device drivers (VxDs).

For additional information on prolog and epilog code, take a look at this Code Project article: Playing with the stack and do additional Google searching for the terms "prolog" and "epilog".

When we use a naked function, we have a few guidelines that we must follow. For a list of guidelines that you must follow, please take a look at this article: Rules and Limitations for Naked Functions. An important thing to remember is that you cannot declare variables inside a naked function. Instead, they must be declared outside of the function. If you place a variable declaration in a naked function, you will be referencing an address on the stack. Aside from that, it is advisable to only place the least amount of non-assembly code in the main codecave function as possible. You should place everything else in the support functions.

Since we are using a codecave in a DLL and using the `CALL` method, we will need a total of two variables and two functions. The first variable will hold the current score obtained from the game. The second variable will hold the return address when we enter and exit our codecave. For this simple example, we will just display the current score to a console window to show that everything works.

What follows now is the relevant code for the codecave implementation. The `DllMain` and extra utility functions are not shown.

Hide Shrink Copy Code

```
// This variable holds our current score
DWORD currentScore = 0;

// This variable holds the return address, it must be global!
DWORD ExtractScoreRetAddr = 0;
```

```
// This is our higher level C++ function that is called to display
// the current score
void DisplayCurrentScore()
{
    // Simply display the current score to the console
    printf("Current score: %i\n", currentScore);
}


// This is our codecave function, we must remember to
// make it a "__declspec(naked)" function
__declspec(naked) void CC_ExtractScore(void)
{
    __asm
    {
        // The first thing we must do in our codecave is save
        // the return address from the top of the stack
        pop ExtractScoreRetAddr

        // Since we know the current score is in EDX, copy it over into
        // our variable
        MOV currentScore, EDX

        // Remember that we need to preserve registers and the stack!
        PUSHAD
        PUSHFD
    }

    // Invoke our C++ function now
    DisplayCurrentScore();

    __asm
    {
        // Restore everything to how it was before
        POPFD
        POPAD

        // This is an important part here, we must execute whatever
        // code we took out for the codecave.
        // Also note that we have to use 0x3B9ACA00 for a HEX #
        // and not 3B9ACA00, which would be misinterpreted by the compiler.
        CMP EDX, 0x3B9ACA00

        // The last thing we must do in our codecave is push
        // the return address back onto the stack and then RET back
        push ExtractScoreRetAddr
        ret
    }
}

// Initialize function called by the loader's inject function
extern "C" __declspec(dllexport) void Initialize()
{
    // We will place a codecave at the address 0x01017580.
    // The function will call CC_ExtractScore
    // and one extra byte will be NOP'ed
    Codecave(0x01017580, CC_ExtractScore, 1);

    // Create a console since we are in a DLL
    CreateConsole();
}
```

That is not so bad now, is it? Remember that only the codecave related code is shown, the rest of the code is part of the project. If we recall all of the theory we learned in the second section of this article, everything seems to be here. We first have our codecave save the return address to a variable. That is seen in the line `pop ExtractScoreRetAddr`. Next, we save our data to another variable as seen in the line `MOV currentScore, EDX`. Before we call our support function, we save the registers and flags to the stack and then restore them later. We finally execute the code that was taken out for the codecave, which was the line `CMP EDX, 0x3B9ACA00`, and return to where we are supposed to be using the stored return address.

The `Initialize` function is an exported function that our Loader will use to inject the DLL into the Pinball game so the EXE will call the codecave. The function `Codecave` will do most of the work for us in writing out the codecave itself. Since we are using

the CALL method to get into the codecave, the Codecave function creates a CALL instruction using the address of the function that we pass in. The last parameter is the NOP count that we specify to erase extra bytes that need to be taken out. This number is simply the total bytes you want to codecave minus 5. In this case, there are 6 bytes in the CMP EDX, 3B9ACA00 instruction as we can see from the assembly listing. Five of these bytes are for the codecave, so that leaves on extra byte we must NOP so the program does not crash on resume due to an invalid byte sequence. If there were exactly 5 bytes, we would not need any NOPs. If there were 7 bytes we would need 2 NOPs.

To create future codecave DLLs yourself, you can use the provided project as a template. You will have to update the Initialize function as well as implement new codecaves for yourself. Now that we have the Codecave DLL completed, it is time to move to the last step in this process, putting it all together!

Step 3: Putting it all Together

In order to see the final product of what we have just made in action, we must find a way to get our DLL into the Pinball process. Once we do that, the Initialize function has to be called to make the DLL patch the program so the codecave is called. After that, whenever the score is updated, our codecave will be triggered and we will see our score being displayed to the console.

To accomplish this last step, I will refer to and use a previous article I have written: A More Complete DLL Injection Solution Using CreateRemoteThread. I will use that project to create the Loader to inject our DLL into the process. The loader itself is pretty simple; the WinMainshown at the end of that article has only been modified slightly for this one:

Hide Shrink Copy Code

```
// Program entry point
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                   LPTSTR lpCmdLine, int nCmdShow)
{
    // Structures for creating the process
    STARTUPINFO si = {0};
    PROCESS_INFORMATION pi = {0};
    BOOL result = FALSE;

    // Strings for creating the program
    char exeString[MAX_PATH + 1] = {0};
    char workingDir[MAX_PATH + 1] = {0};

    // Holds where the DLL should be
    char dllPath[MAX_PATH + 1] = {0};

    // Get the current directory
    GetCurrentDirectory(MAX_PATH, workingDir);

    // Build the full path to the EXE
    _snprintf(exeString, MAX_PATH, "\"%s\\PINBALL.EXE\" -quick", workingDir);

    // Set the static path of where the Inject DLL is, hardcoded for a demo
    _snprintf(dllPath, MAX_PATH, "PinballCodecave.dll");

    // Need to set this for the structure
    si.cb = sizeof(STARTUPINFO);

    // Try to load our process
    result = CreateProcess(NULL, exeString, NULL, NULL, FALSE,
                           CREATE_SUSPENDED, NULL, workingDir, &si, p);
    if(!result)
    {
        MessageBox(0, "Process could not be loaded!", "Error", MB_ICONERROR);
        return -1;
    }

    // Inject the DLL, the export function is named 'Initialize'
    Inject(pi.hProcess, dllPath, "Initialize");

    // Resume process execution
    ResumeThread(pi.hThread);

    // Standard return
    return 0;
}
```

Remember that only the `WinMain` is shown above, the rest of the code is in the project file itself. At this point we now have all of the main pieces. It is time to test! Copy over the "*PinballCodecave.dll*" and "*PinballLoader.exe*" files into your Pinball folder, which is "*C:\Program Files\Windows NT\Pinball*" by default. Run the "*PinballLoader.exe*" file to start up the Pinball game, you should see a DOS console popup as well. If everything works out well, you should be able to play the game. When you score points, your current score should be outputted to the console window. Neat, huh? If anything goes wrong, take a look over the address you are placing the codecave on as well as all of the minor details. The good thing about this low level stuff is if you mess up, 90% of the time you will be aware of it since the program crashes or does something unexpected.

Conclusion

It has been a long and challenging journey, but you have finally reached the end. At this point, you should have gained a basic, but complete, understanding of what a codecave is. We have seen through the theory section of what a codecave does and how it can be used. You now know three important attributes of codecave that you have to keep in mind when designing your own. They are the location of the codecave, the entry and exit points, and the stack and register preservation techniques. You have a practical example to reference as well as a template code at your disposal for your future projects. You might have even gained a little knowledge of a few new tools that you can use in your future endeavors.

The big question left that always comes at any end is "what now?" From here, you can continue to explore using codecave to accomplish various tasks that you might not have been able to do before. Sometimes it is challenging to figure out "what to do", so if you do not have anything to work on immediately, do not worry about it! Just remember what you have learnt and perhaps you can apply it someplace else in the future. I hope you have enjoyed this article, I know it is very long but I hope to hear your feedback.