# ECE 6747 Advanced Topics in Malware Analysis

# MODULE 4 High Level Language Constructs in Assembly

# TRANSCRIPTS

## L1- Function calls, If/then/else, Switch

>> Hello everyone and welcome back to Advanced Topics in Malware Analysis. In this lesson, we're gonna be learning how to identify high level language constructs after they've been compiled to assembly language. To give you an overview of what you're in for, we're gonna learn how to interpret high level language structures once they've been compiled to assembly.

We're going to learn how to identify function calls and reconstruct different switch statements, loops and other compiler induced branching operations. We're going to figure out how to discover data structures, like arrays, or data structures that build linked lists in assembly language. And finally, we're going to learn how to put all these things together and build back up the high level language structures that we're used to when we're looking at assembly language code.

The key thing to understand is how those high level language constructs look once they've been compiled into assembly language. Functions, specifically the calls to a function or returns from a function, have already been discussed in earlier parts of these lessons. So you're pretty familiar with identifying where functions begin and end, and how you can call a function.

Control structures, such as loops, branches, if/then/else statements, or switch statements, are gonna be covered in this lesson. Later on, we're going to talk about data structures like structs/unions, arrays and even linked lists and using pointers to connect data structures. All of these things are going to become apparent to you once you know what to look for in the assembly language.

Here's an example of IDA automatically annotating function calls for you and their arguments. This can be reverse engineered by looking at the arguments that get pushed onto the stack. IDA has a set of built in signatures for each function in known libraries that reverse engineers might run into.

These are called flirt signatures, and you can Google these to find more information on how they work. At a high level, they search backwards on the stack or in the registers to identify the arguments given to a function call. So in the example shown here, we're calling the connect function.

And IDA pro automatically knows that connect requires three arguments, the buffer, the name and the name length. The pointer to the socket buffer is the first argument, and so IDA can reverse engineer where on the stack that should be, and find the push instruction that pushed that argument onto the stack.

This allows IDA to automatically add a comment, helping you as the reverse engineer identify those arguments. However, the more complex a binary gets, such as moving those arguments into registers and then moving them onto the stack with complex move operations, starts to trip IDA's analysis. Notice that in the second case shown here, IDA can still identify the connect call, but it can't identify where the arguments are being put on the stack.

This is where you, as a reverse engineer, are gonna have to take over and add comments on your own. If/then/else statements look pretty much like you would expect them to. The typical assembly sequence is that you're going to compare the value of some register or memory expression with a constant or another register.

There's then gonna be some conditional jumps that jump you to the if block, the else block, and then there's gonna be a label where you drop to after the if/then/else. The CPU can even use special flags to make this happen in an optimized way. Switch statements get a little tricky, because there's multiple ways that a compiler can choose to implement a switch statement.

The most straightforward way, and what's chosen by most compilers as often as possible, is a table implementation. The compiler is gonna put together a list of all the possible switch cases that you could jump to, based on the switch value that you're using. The compiler is then gonna build a table of addresses that should be jumped to.

And it's going to use that switch variable as an index into the table. We're going to see how that works in the next slides. The problem here is that you have to be able to use that switch value as an index into your table. So if you have very large breaks between the different values in your cases, the compiler has to change to a tree implementation.

In this case, the compiler is going to embed a binary search tree into the executable itself to match the case that you're looking for at runtime. Here's the idea behind the table implementation of a switch case. Notice here that we're switching on the value A, and A can take a value between 1 and 4.

We also have a default case to go to for values outside of that. The first thing the compiler is going to do, is set up all the different code for each of the cases in a block, so that you can jump to each one of the cases. So if we have our case 1, we'll have a corresponding case 1 code, that allows execution to jump to the case 1 code when the value of a is equal to 1.

The compiler also insert jumps to jump you out of the block at the end of each case. Notice case 3 does not have a break statement, and so it does not get a jump. Its execution flows directly into case 4, as you would expect, given a case without a break.

Next, the compiler sets up a jump table, that's all of the addresses of these different blocks of code that need to be jumped to at runtime based on the value of a. The compiler then does a little bit of math to translate a into an index into the jump table, so that at runtime you can use the value of a to index into your jump table, and then execute the unconditional jump at the end of that block to jump execution into the case corresponding to the right value of a.

Together, these build up the way that a switch statement works in an assembly program. Now that's compiled without any optimization. So here's an example of real code compiled without any optimization and turned into a real binary switch statement. Originally here we're switching on the value of a, and we're printing different strings based on whatever values a takes.

If we compile this with O0, that is no optimization, we get the following assembly instructions. First, we can see that all of the strings that get printed get put into the cstring section of the binary. Looking at the switch statement's code, we can see that we start off calling scanf to read a value for a.

The value of a is stored in the eax register and compared against 3. The reason to compare it against 3 is to check if we need to jump off first to that default case. That's the jump above instruction that's highlighted on the slide. If we hit that default case, execution is going to jump down to the implementation of the default case, which I'm going to show on the next slide.

Here's that default case where we can see we're loading the no idea string into rax and calling the put s instruction. This is going to print No idea, and then fall straight through to the end of this main function and the return. Returning back to the switch statement, if we did not jump off to the default case, we're going to immediately load the value of the jump table for the other cases into the rax register.

If we take a look at that jump table, we can see that it's four offsets, stored one after the other in memory aligned by four bytes. The aligned by four here is important, because that's the size of each pointer. And each one of these values in the table is going to point to one of those cases that we're trying to jump to.

And you can see we can index into this table, at offsets 0, 1, 2 and 3, to grab the different pointers to the case statements. So once we've loaded the offset of that jump table into rax, the binary is going to perform a little bit of math to get that index into the table from the value given for a.

At the end of that math, we're going to load that final jump target into the rax register, and we're going to execute a jump rax shown at the bottom of this screen. Here's the implementations of those four different cases. We can see the 0 case at the top, where we're printing 0, then the case 1, then the case 2, and finally the case 3.

Following the case 3 is that same no idea default case that we saw before, and the return from the function. You can also see the new label added just before the function return, which is where all of the other cases will jump to after they finish executing. And finally down at the bottom, you can see that's where the compiler has put that jump table that we saw before.

Now that you see all of the labels in line in this code, you can compute the concrete value that those offsets are going to take in the jump table. Things look a lot different when we compile that same code with -02. 02 is gonna add a lot of optimization, and the entirety of the code can fit on one screen.

At the beginning, we're just gonna immediately read the value of a, using scanf, and then use that straight away to jump to the default case like we did before. The cases have been condensed, here, and you can see the implementation of each one is just loading the string into the rdi register.

That then jumps to this final location that calls put s and then returns from the function. There's no need to duplicate the call to put s throughout the code, and these additional levels of optimization realized that, and brought it all into just one final move before we return from the function.

You can still see the jump table aligned by four down at the bottom of this slide. There's no way to avoid that in this code. A jump table only works if you can easily use the value of the switch as the index into the jump table. If there's huge differences between the values of the cases, you can't get away with using a jump table, because that index computation would become too difficult to do with all the huge gaps in the jump table.

Now sometimes compilers will get around this by adding empty space in the jump table, but that's a little dangerous, because it allows for function pointers to be injected into the binary. And if those spaces have to become too big, such as the example shown in this slide, a compiler will choose to implement the switch case as a binary search tree embedded in the code.

You can see how this works from the example. The compiler will find the middle case that's defined in the code, and then begin splitting the range of the cases in half with every comparison. So in this case, the compiler will first check if the value is above 1,100, if it's above 1,100, it will proceed to the above 1,100 branch of the tree, and so forth and so on.

Each comparison allows the compiler to knock out half of the cases that it would have to check. You'll see these embedded in a number of binaries with switch statements that grow very large. So keep your eye out for them when you're reverse engineering.

## L2- Loops

>> Hello everyone, and welcome back to advanced topics in malware analysis. In this lesson, we're gonna continue talking about how to identify high level language constructs, once they're compiled into assembly language. This lesson is all about loops. Loops are very much what you would expect. When you write a loop in a high level language like C, you end up having a pre-tested loop that checks for some condition, and then the body of the loop that repeats until that condition is met.

In assembly, you often have compilers that will change pre-tested loops to post-tested loops. So in this example, we can see that the c code originally is while c is less than 1000, add c to some array and increment c. If we compile that to assembly language, you should be familiar with the instructions that we see here.

The compiler will add a label for the top of the loop, and then perform the loop body. Use a compare instruction to check the loop condition, and then jump with a conditional jump, back to the label at the beginning of the loop. Loops can often be optimized by the compiler to use what's called the rep instruction.

Rep is actually a special instruction prefix, that goes before other instructions. Rep tells the processor to repeat the instruction that follows ECX number of times, or until the zero flag equals 0 or 1 depending on the condition. If you see the table in this slide, it explains the different combination of the rep instruction prefix, and what termination conditions will stop the repetition.

This is used to repeat loops that consist of only one instruction. That way, the processor doesn't need to process additional instructions. It can just stay on that single instruction executing, until the termination condition is met. There's also a loop instruction that's also used by X86 processors. But this is very rarely seen, and so you can look it up if you ever see it when reverse engineering.

A concrete example of where the rep instruction prefix is used, is actually in libc's memcpy. If you reverse engineer the implementation of memcpy, you'll see towards the bottom of this slide, that the memory is actually moved via a single instruction, the move sd instruction with a rep prefix.

And the size of memory that you're moving is stored in ecx. Loop unrolling is another common trick that compilers use to optimize loop bodies. Duplicating the body of the loop, reduces the number of jumps that the processor needs to execute. So what compilers will do, is eliminate as much branching as possible.

It will increase the code size, but it will reduce the number of jumps the processor needs to implement. You can instruct gcc, to try its best to unroll loops with the funroll-loops command. So for this example code, I've compiled it on 64-bit Windows using the gcc command shown.

The output assembly language is on this slide as well. And you can see that instead of moving only one character at a time, the compiler has actually unrolled the loop body to move eight characters before testing rax, and jumping to the head of the loop. In an extreme case, compilers may even completely unroll the loop so that there's no loop at all.

If we compile that same C code with a higher level of optimization, we can see that the compiler actually removed the loop entirely. It was able to figure out that you're actually moving 64 characters, and instead, optimize the use of registers to just fetch 64 characters one after another, and move them into the destination.

So if you look at the assembly language here, there's actually no loop at all, where the original source code had a loop. This doesn't make reverse engineering any easier. But at least you'll be aware that when you see large memory moves like this, it could have been in a loop originally, and just optimized out by the compiler.

## L3- L1Arrays

>> Welcome back everyone. In this lesson, we're gonna identify arrays once they've been compiled into a binary. Arrays are generally gonna be indexed by another variable, and can easily be located in assembly code by finding a base pointer to the beginning of the array and then the index from that base pointer that you're trying to access.

Sometimes it's even possible to determine the length of the array by closely analyzing the code. Arrays that are constant, that is defined as a constant array in the code itself are gonna be harder to find because the compiler can compute ahead of time where those arrays are gonna be in memory.

And so it can remove the index plus base computation. We'll see an example of this later on. Accesses to dynamically allocated arrays are usually much easier to deduce. This is because the compiler doesn't know where that array is gonna be placed in memory ahead of time. And so, you're going to see that same base plus offset computation.

If you're using a tool like IDA or Ghidrah to help your reverse engineering, these tools are smart enough to try to identify arrays when they can find these patterns in the binary. Sometimes it can miss these, especially for arrays of unions of different data types. So you may have to go in and instruct the tool on where the array is and what the size of each element is.

For IDA, this is as simple as right clicking on the data and telling IDA to convert it to an array. You can set the size of the array, how many items you think are in the array, and the width of each item. If you're following along and Chris Eagle's the IDA pro book, this is gonna be described in chapter eight.

In Ghidrah, it's the same thing except a few more steps. Ghidrah also tries to identify arrays. But when you need to tell Ghidrah to understand a specific piece of data as an array, you right-click on that data, go to the data menu, and then create array. You can set the size of the array and bytes, and you can change the different data types inside of the array.

Here's an example of a globally defined array in some C code getting compiled into assembly language. You can see that the compiler can easily figure out where the elements in this array are gonna be placed once the compiled binary is finished. This allows the compiler to do some optimizations and replace the calculation of global array at offset zero or global array at offset one with direct pointers to the memory where those values are.

If you look at the axis's denoted by one, two and three in the disassembly, you'll see that instead of computing the base of the array plus the index, the compiler just simply replaced it with direct pointers to those fields. The access denoted by four however, the compiler did not optimize.

And you can see that the compiler is using the base of the array plus some index, in this case, eax times four, to get to that fourth element in the array. Arrays that are dynamically allocated on the heap will be much easier to identify. As we've seen before, you can usually find the call to malloc or other allocation functions in the binary.

This will even give you a hint as to the size of the array. In this case, we can see that we're allocating three times size of int elements in this array. And so that translates to push zero C hex and then a call to malloc, denoted by number five.

Later on, the compiler cannot pre-compute the address of the fields of this array. The compiler is forced to use the base plus index to access the fields of this array. You can see in numbers one through four, where each field of the array is accessed via a computation of the base pointer to the array plus the index into the array.

## L4- Struct/union analysis

>> Hello everyone, and welcome back to Advanced Topics and Malware Analysis. Anyone who's programmed in C knows that you have to use structs or structures as often as possible, and unions, as well. Both of these constructs make reverse engineering programs a little bit more tricky. So we're gonna talk about how to handle them in this lesson.

What is so difficult about reverse engineering structures and unions are that field names and other helpful debug symbols are usually lost. Malware definitely are not going to reveal their debug symbols to you. And they're just gonna be replaced by raw offsets by the compiler. Global or statically allocated structs and unions are particularly difficult because the compiler can get away with reducing as much of the math as possible to get to the fields inside of the structure or the union.

And often, you'll only see a constant address being used. Similar to arrays, if the structure or the union is dynamically allocated, you'll see the memory allocation function. And this can help you identify a structure or a union. It can be very complicated to reverse engineer structures and unions, especially when there's alignment issues present.

And we'll see an example of that later in these slides. But good news for you reverse engineers out there, common structures like sockets or other things that are used by many applications, are pre-baked into IDA Pro and Ghidra to help automatically identify those structures. If a particular structure you're working with isn't automatically identified, you can always add it manually to IDA or to Ghidra.

Here's an example of how IDA automatically identifies structures. IDA actually has a tab called the Structures tab, where you can go in and add new structure definitions or edit existing structure definitions to help with reverse engineering. Ghidra has a very similar function. In Ghidra, structs and unions are actually handled much cleaner than in IDA.

This is one plus of using Ghidra for reverse engineering. The functionality is actually very useful when it's combined with Ghidra's built-in decompiler. When Ghidra detects a class, you can actually define each of the members in memory. That makes the decompiled code significantly more readable because the decompiler will assign the regular human readable names to each of the struct members.

Be careful of alignment issues that are introduced by the compiler. Structures that are defined in source code only provide field names and types. The compiler is free to lay out those fields however it wants in memory, and different compiler flags will choose different field alignments. For example, a memory-efficient compilation will choose a minimal size in memory for that structure, also referred to as a packed structure.

This isn't gonna be optimal for processor caches, though. So if the compiler is instructed to be processor-cache-efficient, the compiler may choose to lay out fields on a power of 2. If we look at the example shown on the slide, the same structure written in C only provides the field types and names.

In this case, fields 1 through 5, that are an integer, a short, a character, another integer and a double. Looking at the size of each of those fields, we noticed that the character is the smallest field at only one byte in size. If we're going for a packed data structure, that is, the compiler is trying to minimize the size in memory, it can lay out field 4 at the byte immediately after field 3.

That is, starting field 4 on byte 7 of the structure. However, if the compiler is instructed to be processor-cache-efficient, it could lay these fields out very differently. It could actually skip the additional byte after field 3 and layout field 4 on the eighth byte of this data structure.

This gonna inflate the final size of the data structure to 24 bytes in size, when the minimum size could have actually been 19 bytes. These are just two examples, but there's actually a number of different compiler combinations that could result in very different structure alignments in memory. So just be aware when you're finally reverse engineering a structure and you realize that there are fields that are either not being accessed or being accessed in a strange way, it might have been the compiler doing some optimizations.

This can be quite confusing to reverse engineers, especially when there is extra space added to a structure. You're not sure if that extra space is actually another field that you haven't seen used yet or if it's just some extra space added by the compiler. Keep these issues in mind when reversing tricky binaries that might have handwritten structures in them.

Similar to globally-defined arrays, globally-defined structures can be accessed in code just by the name of the structure and the name of its field. But when that's compiled into a binary, the compiler is free to strip away all of those indices and compute constant addresses to be used in the binary.

We can see this with the example on-screen, where the structure defined on the previous slide is being used by the main function. When that's compiled into assembly language, you actually don't see any computation of where field1, field2 or field3 are. The compiler has just put direct pointers to those fields into the code.

A heap-allocated structure is a little bit easier to identify. Again, you'll see the call to malloc and the push, or the argument to malloc, with the size of that data structure. Fields within that data structure then follow the same base plus offset computation. So for instructions marked with 1, 2, 3, and 4, you can see the different fields of the data structure being accessed using the base of the data structure plus the index into that data structure, or the number of bytes into that data structure of each field.

If you have to roll back to look at the same definition of this data structure that we introduced previously, feel free. Here is that same data structure compiled with different compiler flags that led to an entirely different alignment. You can see that where the previous code we were looking at was accessing that field at eax + 8, with this alignment, that same field is being accessed with eax+7.

This is that case we talked about before, where the compiler can choose different alignments for the same source code of the data structure. Keep an eye out for this when you're reverse engineering in the real world.

## L5- Arrays of structs

>> Hello everyone, and welcome back. We've talked about a lot of the constructs that you would see, when writing in C code, once they've been compiled down to a binary. In this lesson, we're gonna get a little bit more complex with arrays of structs, more advanced data structures, and a few other tricks, that malware will throw at you.

Recall the structure alignment problem that we saw in the past slides. Just look at the definition of this data structure, because we're gonna use it in the next slide. In this code, we're allocating an array of those data structures. You can see the call to malloc with the size of that data structure times five.

So we're gonna to end up with an array of five of these data structures. The code, then do marked by line one, shows an access into that array at index idx, and then using field one of that array. When this is translated into assembly language, you can see that the compiler has optimized the call to malloc to just push the full size of all five of those data structures.

So you can see we're going to malloc 120 bytes. The compiler has optimized the call to malloc to just use the final total number of bytes that we want to allocate. The instructions marked by three and four are performing the index into that array. And then the index into that structure within the array.

You'll see this common pattern when a binary has arrays of structs. It's gonna be multiple levels of indirection based on a single base pointer, giving you another base pointer, until you finally index off of that base pointer. I know it seems a little confusing, but when you're reverse engineering a binary like this, keep a piece of paper in hand, so that you can scribble down your ideas for how the data structures are laying out in memory.

This will help you tremendously as you go about reverse engineering more tricky malware. The more complex the data structures get, the more essential it's gonna be for you to keep a piece of paper around, and keep track of how the data is being used by the binary. This is an example of a linked list.

If you look at the C code on the side, you see we've defined a struct called node, with a pointer to the next node, and an integer storing the data. The print function below that, takes one of those nodes as an argument, walks down the linked list, and prints each of the integers.

When we translate this into assembly code, we can see that the print function is gonna take that first argument in the RCX register. That's a pointer to the first node in this linked list. Pause the video and take a minute to reverse engineer the assembly code on the slide, so that you can follow along with how the linked list is being used.

We can see that the pointer to the next element of the linked list is being stored in the RBX register. The test RBX RBX instruction, is actually computing the pointer not equal null check that keeps this loop rolling. Inside of the loop, we can see that RBX is being used as the base pointer to the beginning of that node structure.

This was compiled on 64 bit Windows. So the pointer to the next node is 8 bytes in size. That means, to access the integer inside the node, we have to compute RBX plus eight. Right after the .L8 label, we can see that the code is moving that integer into edx, and then calling print F to print the integer that's in that node.

The loop continues until it finds a node whose next pointer is null. Given how tricky that simple linked list example is, you can imagine how more complex data structures can get to be quite a headache for reverse engineers, trees, heaps, heaps of trees. These things start to build up with more and more complex applications.

You might run into trouble reverse engineering data structures, and just fall back to looking at the code itself, or even the system calls, or library calls in the binary. To get a better understanding at a high level of what's going on, before trying to dig down into how each piece of memory is being used.

It may not be time efficient to reverse engineer the most complex of data structures. Interestingly, languages like C++ or other object oriented languages, actually have very rich runtime systems. Objective C is the best example of this. in Objective C and other high level languages with runtime type systems, you sometimes can reverse engineer full definitions of the data structures, in order to better understand how they lay out.

If you're curious about this, there are several books at the end of this module with pointers to where you can find out how to reverse engineer Objective C, or c++ a little bit better. It can get much more difficult, and eventually you might just give up, and assume a structure is just a blob of data.

And fall back to a higher level understanding like library calls of how a malware executes. The best case scenario is malware that's always written in assembly language. These malware rarely have complex data structures, because they would have to be coded entirely by hand. Unfortunately, malware authors like high level languages as much as we do, and so you won't run into these types of malware too often.

## L6- Importing shared functions

>> Hello everyone and welcome back to Advanced Topics in Malware Analysis. In this lesson, we're gonna talk about how binary programs import shared functions from external libraries. Importing shared functions can be done either statically or dynamically depending on how the code was compiled. Statically compiled code is a lot easier to reverse engineer because you have the entire library function baked in to the single binary that you are analyzing.

Dynamically linked code can be a lot more challenging to reverse engineer if you don't have access, to the binary of the library. That's because the connections from the binary, to the library function, aren't made, until runtime, when the binary is actually being executed. An additional program called the loader, is gonna step in and patch the function addresses at runtime, to connect dynamically linked libraries.

Different executable formats, follow different conventions for importing external libraries and so we're gonna cover that in these slides. On Linux or Linux derivatives, ELF shared libraries use a combination of a PLT and a GOT. The GOT or global offset table contains pointers to symbols that are imported from other shared libraries.

Your binary is gonna contain a global offset table and that global offset table is gonna be used to connect your binary to the external functions that it needs to call. The job of doing this connection falls to the PLT or the Procedure Linkage Table. The Procedure Linkage Table contains code that transfers control from your binary into the shared library that you're trying to call.

This works by fetching the pointer that's stored in the global offset table and then jumping there. If we look at the diagram on the right hand side of the screen, we can see that libdl.so is gonna make a call to calloc. Calloc the function resides in the libc library so what the compiler is gonna do is insert a call to the calloc entry in the PLT of libdl.so.

The PLT entry is gonna be just a single instruction, that jump instruction, which is gonna fetch a pointer to the callback function at runtime. That pointer is stored in the global offset table. So you can see following arrow number 1, that's gonna be the call into the PLT.

Then following arrow number 2, that's the fetch to get the pointer to the calloc function in the GOT. Then arrow number 3 is the jump from the PLT to that pointer that it was fetched from the GOT, landing the execution at the beginning of the calloc function. Of course, when you first start executing, you don't know where the calloc function lives in memory.

So all of those GOT entries are initialised to point to a function in the loader that will resolve those symbols on the fly. Before the first time you call an external function all of the GOT entries are gonna point to the loader. The first time you call that external function, that PLT entry is gonna jump you to the loader instead of the function that you think you're calling.

The loader is gonna look up where in memory that function you're trying to call lives, patch the address that's stored in the GOT, and then transition the execution to that function. Then all subsequent calls to that external function can just fetch the pointer that's been patched in the GOT and go directly there during execution.

When you're reverse engineering with a tool like IDA or Ghidra, you can see this by double clicking on calls to external functions. So for example, if we double clicked on the call print f instruction, it's actually gonna bring you to the PLT entry for print f. You'll recognize that jump, and then a data fetch for the address that you're jumping to.

If you double click on the address of the data that's being fetched, it will bring you to the GOT in that binary and you'll see that the GOT has an entry for a pointer to where the print f function is. On Windows PE executable files, use a combination of an import address table and the idata section.

The idata section of the PE file contains two separate tables, one called the Import Lookup Table, and one called the Import Address Table. The Import Lookup Table is an array of structures that contain a pointer to the name of the function that you're trying to import. Those structures in the Import Lookup Table Mirror structures that are in the import address table.

Basically, when you begin executing these two tables will be identical, the same array of structures with the same pointers to the names of the functions when you call an external library The windows loader is gonna patch the address in the import address table to point to the runtime address of that function.

The Import Lookup Table stays constant throughout execution. The windows loader unlike Linux will try to be proactive and may load some external DLLs ahead of time and patch that Import Address Table right as soon as you begin execution. This actually depends on how the external dependency was declared in the source code and it would allow the compiler to do some optimizations.

Now we'll take a look at that. This code is a case where the compiler could do the optimizations, that means in the original source code, the call to the socket function was actually declared as an external function that needed to be imported. Similar to ELF all external symbols are gonna need to be initialized by the loader at runtime.

In Windows, the loader is actually a helper function that hands off control to the operating system kernel. In Windows, the loader lives entirely in the operating system kernel. If the DLL has been proactively loaded, then the loader is just gonna link the two functions together by updating that function pointer.

And you can see in the code shown below that you actually only have a single step where the call to socket to the socket function has been compiled to simply a call RAX and above that is a move RAX with a pointer to the socket function. That pointer of course is in the idata section where the loader can store the pointer to the socket function when it does it's linking.

In the case of the original source code did not declare the included function as an external dependency The compiler didn't know to do that little optimization. In this case, a stub function is gonna be created, often called a thunk. That will perform a similar jump to a pointer to the external function that you're trying to call.

This operation is very similar to how ELFs PLT works, except in this case the stub function lives in the text section of the binary and the pointer to the external function is still in the idata section. For an excellent additional reading on how loading and linking work on both Windows and Linux, you should check out these links in this slide.

These are some blog posts by researchers at Symantec, the company that makes Norton AntiVirus. The researchers here reverse engineered the linking and loading procedures on both Linux and Windows and did an excellent job documenting their findings. And that brings us to the end of our lesson. To wrap up, we've looked at how high level structures get compiled into assembly, how to identify those structures when you're reverse engineering and we've looked at how different data structures and arrays and even importing external functions work when you're seeing them compiled and in a binary that you're reverse engineering.