# ECE 6747 Advanced Topics in Malware Analysis

# MODULE 1 TRANSCRIPTS

## L2 Assembly Language

>> Hello everyone and welcome back to advanced topics on Malware Analysis. Today we're gonna be covering an Introduction to Assembly Language. For this slide deck, I hope that everyone can learn to recognize and define x86 assembly language. And that's gonna be very important because we're gonna need to distinguish different registers and different executable files.

How Intel and AT&T syntax can be different on different platforms. And we're gonna learn how the stack and then heap or managed when you allocate memory. And all this is going to be important for understanding how malware and any binary software operate. So like I said before, malware do not come with source code.

Now, they're written in high level languages. Just like any program, now we are written in something like C, C++, even Python, sometimes. And a compiler will take in that high level language, chew it up, perform whatever magic it's going to, and spit out a binary executable. Literally the ones and zeros that execute on the processor itself.

And since that's all we're left with, we're going to be looking at that binary code, to understand what the malware was trying to do. Is this hard? Yes, it's very difficult, but there are some tools of the trade that make this possible. You see, an executing program is just a sequence of 1s and 0s that the CPU understands to be instructions.

Reverse-engineering, therefore, is the process of analyzing a subject binary program to create representations of the program's logic at a higher level of abstraction. So basically pulling those 1s and 0s back up to the high-level code that everyone can understand what it's doing. Luckily for you, binary programs can actually be disassembled back into assembly language.

You see that sequence of 1s and 0s that the CPU choose up and understands as instructions are actually a one to one mapping with assembly code or a sequence of mnemonics that represent the processor instructions. We call each of these Opcodes. So I can give you an example.

This sequence of ones and zeros 0000 01 and so on actually represents the add instruction. It's telling the CPU, add 10 the AL register. And we can even parse that a little bit further, and the CPU knows that that first 2 bytes of instructions can be chewed up.

And that represents the add to the AL part. And then the next part after that opcode is the operand that's that 10 in binary. So essentially, assembly language is gonna become your new hobby. You're gonna need to learn everything you can about Intel assembly. There's a number of ways to be successful at this.

Certainly read as much as you can find about assembly language and assembly programming. The best resource for this are the Intel architecture manuals. Intel has published manuals on every single instruction that they're processors understand and how they work and what they do. You can also write native assembly applications and verify that they work properly.

You could compile programs in a language that you're familiar with, and then execute those and examine them in assembly language to understand on what's going on. But it all comes back down to read, read, read, read, nothing is going to replace your brain for understanding assembly language. This video is gonna give you a quick tutorial on assembly language, because there's so much to learn there, we're just gonna be able to scratch the surface, and a lot of this is gonna be learning as we go.

So here I'll give you an example. Here's some C code for hello world that everyone is familiar with. Now, if you look at the C code, how much code is gonna be generated when you compile it? How complex is that executable gonna be? So if we use that GCC command listed on the slide to compile it, here's the assembly language that we're gonna be looking at.

And I know this looks like gibberish to everyone right now but by the end of these slides, you're gonna be able to understand every line of this assembly code and it's gonna make perfect sense to you, I promise, hang in there with me. So to begin to understand Intel assembly, you need to start thinking like an Intel processor.

Here's an example of a basic 16 bit CPU that chugs through instructions and operates just like the modern 64 bit processors of today. You can see there's a bank of registers. These registers, store the live data that the CPU is interacting with. There's also a special flags register that controls how the CPU makes decisions.

There's a pipeline of instructions that are being queued up that tell the CPU what to do. And then there's different data formats that explain how data can be stored and moved to and from RAM and that happens via the stack and via the heap. And we are gonna go through each of these in detail as we get through this next module.

## L3 Basics of Assembly Code

>> Hello everyone, and welcome back to Advanced Topics and Malware Analysis. This slide deck is gonna cover the basics of assembly code and registers, basically how the CPU handles data and what it does with it. To understand registers, you've got to think of how the CPU actually stores the data that it's working with.

Back in the old days, of the 8088, 8086 processors, you only had 16 bit registers, and those were tiny little banks of numbers that could only store 16 bit digits. That quickly became insufficient, so new processors like the Pentium family introduced 32 bit registers. That doubled the size of the bits that you had to store numbers in.

And today, everyone is super familiar with 64-bit register that up until now have pretty much been ubiquitous in computers that we deal with. But you are gonna need to understand the differences between 16-bit, 32-bit and 64-bit processors. Because the register names will be quite different, the feature sets will be quite different, and malware in the wild will abuse these differences.

In order to make it difficult to reverse engineer what they are doing. So when I'm talking about registers, there is a certain set of registers that everyone needs to know. There is a certain grouping of registers that are called general-purpose registers. These are EAX, EBX, ECX, EDX, EDI and ESI.

These are gonna be used for specific purposes, a sort of a rule of thumb. EAX for example is the accumulator. So if you are doing additions, that normally would be done against the EAX register. Or ECX is the count register. So if you are doing a loop index for instance where you are counting how many iterations the loop has gone through, that'll be in the ECX register.

These are not guaranteed to always hold, but they're definitely rules of thumb that most compilers will try to follow. Which means that malware will often abuse these rules to make it trickier to understand what they're doing. There's also two registers that are specific for keeping track of the stack.

We're gonna talk about how that happens later on in this course. But those are the EBP register, that's the stack base pointer, and the ESP register, which is the stack pointer. These general purpose registers on a 32-bit processor hold exactly 32 bits. So you can store any number that will fit in 32 bits that the program may need while its operating.

So if you consider some giant number like this 775 million, if you try to store that in the EAX register, the processor is actually going to set the bits of the 32-bit EAX register to the binary representation of that large number. And this is actually what that 32-bit register would look like with that number stored in it.

On a 64-bit processor, you have 64-bit registers. Those have been renamed to rax, rbx, rcx, and so on. They're actually the same registers as the 32-bit eax, ebx, ecx, they just have an additional 32-bits added to the top of them. And similarly, the 16 bit registers are ax, bx, cx.

Those are just the lower order 16 bits of the larger 32-bit registers that have been added over time. If you go back in the day to where you only had a 16-bit register, ax alone would just be a 16-bit, so there would be no higher order 32 there.

And then you have a further division of 8-bit registers. These are al, bh, bl, you kind of configure out that these are the lower and higher 8 bits of each of their respective register. So is the higher other 8-bits of the ax register. Al is the lower other 8-bits of the ax register.

This is a helpful diagram for how those registers look in the processor. You have the large bank of 64-bits that make up RAX, inside of that the lower 32 is EAX, and you further divide from there. 64-bit processors also introduce 8 new 64-bit registers. They didn't come up with any creative names for those though, they just call them r8 through r15.

But be on the look out for them in 64-bit code. R8 is 64 bits but there's also the r8d for the lower 32, r8w for the lower 16, and r8b for the lower 8. There is no high or low for these new 64 bit registers. I guess Intel just decided to drop them.

So, just to explain how this looks, again, if we store that giant 775 million number in EAX. The processor is going to convert that to a binary digit, store the ones and zeros of that in the 32 bit EAX register. If you read this from the full RAX register, it still equals the same number.

It just has a lot of zero bits on the beginning of it. So this is what that would look like inside the processor. Now things get tricky if you start reading the smaller divisions of that register. So if you read just the AX register, you're only gonna see the lower 16 bits of that number, so that's gonna give you a different number entirely.

It's gonna give you 14,000 something something, and that's because you're only looking at those lower 16 bits of the full RAX or EAX register. Things get even weirder when you read just the lower order al register, because that's gonna give you either 179 or -77. Remember, 2's complement for representing negative numbers in binary, you might wanna read up on that because it's gonna be important later on in the course.

Now remember the Eflags register that we talked about before. This is a special bank of bits, where each bit represents a flag that tells the CPU how to do different operations. We'll go over each flag when we need to, as we're looking at them in malware. But this slide just gives you an overview of the most important ones.

You have the direction flag for if string operations read memory from high or low addressees. Or you have the carry flag that tells you if the result of an unsigned operation needed to carry a bit over. So you would need to handle that with additional registers. Yes, there are more registers.

There is a bank of floating point registers called St(0) to ST(7). These are 80 bits so that you can handle floating point numbers. They're organized as a stack and they have control registers to turn them on and off. There's also MMX registers that handle multimedia streaming, this is what makes Netflix run so fast on modern processors.

These are still the same floating point registers, they've just been aliased to MM0 to MM7. And then there's additional streaming extensions on modern processors that add new MMX registers that can be used. These are independent 128 bit registers. Don't worry about memorizing all of these right now. We're gonna look at these one off cases as we see them in malware samples.

But for right now, just be familiar that there are many more registers in modern processors than just the general purpose registers. And yes, there's still more. There's control registers that support different processor features, like debugging and virtualization. We're gonna talk about these later on because a lot of malware will abuse these control registers to turn on or turn off or even misuse different processor features.

So you've gotta be aware of how they work and the Intel manuals are a great resource for how these different processor features work. Now we'll move on to different data types, just like modern processors can really only understand binary digits. They also can only understand different data types that correspond to those digits.

So modern processors understand bytes, basically 8 bits. Then you can have two bytes for a word. You can have two words for a double word, and two double words for a quad word, that goes all the way up to 64 bits in size. These are the only data types that modern processors understand and everything is composed of bytes words, double words, and quad words.

And we'll see just how those are used later on when we talk about moving data back and forth from the processor. Now, instructions, instructions are the most important thing here because that's what's telling the CPU what to do. This is basically the code of assembly language. Instructions will start with an optional label, and then have the instruction opcode itself, and the destination operand.

There's also an optional source operand and then you can leave a comment after each instruction with a semi colon. So here's some examples, here for the first instruction in the listing Is a label that tells you from other places in the code, this instruction can be referenced via the label here.

Then CMP is the Compare Instruction Opcode, that means to compare two things against each other. EBX is the destination operand, and in this case we have a source operand, and that's a hard coded number, x for BEEF. So this may be comparing the value in ebx against the constant number BEEF.

Maybe looking for a magic number. There are other instructions that don't have source operands. For an instance, the push instruction will push the contents of a register on to the stack. That only needs a destination operand, because it's taking the value out of that operand and pushing it onto the stack.

Other instructions like XOR can have a destination and a source that may even be the same register. So x oaring, the ebx register with itself will result in zeroing out the ebx register. We'll go over all of the instructions that you'll be seeing in detail later on in the slides.

And now data, how does the processor actually handle different data depends on what the instruction is doing? So first, you can use data as an immediate value, and that's just a hardcoded constant in the instruction. In this case, you're moving The constant BEEF into rax. So you hard code that hex value BEEF into the instruction, and then the move instruction will move that value as a binary number into the rax register.

You can also address data via the registers that hold it. So this move instruction will actually move the value that's in the ebx register into the eax register. So it's actually gonna take the ones and zeros of the eax register and flip them around so that they equal whatever value is stored in the EBX register.

The next slide we're gonna go over how to move data in and out of memory. So, when moving data in and out of memory, you've got to give the CPU an address. And addresses in memory are like starting at zero and going all the way until four GB or however big your bank of RAM is.

These indices can be computed from different registers and even offsets if you wanna make them more and more complex. This slide is just an introduction to that. It's gonna be confusing at first, but it's gonna get much more familiar as you see them over and over again. So to start off, you're going to address memory with a base address.

And that's just a raw number that's interpreted by the processor as an index into your bank of RAM. So in this case, we may store that number in the EBX register. If you look at the first instruction in the listing, it saying to move the bite pointed to by ebx.

If you look at the first instruction, it's saying to move zero into the bite pointed to by the ebx register. In this case, we've previously stored the address we want to use in memory in the ebx register. So that's like a regular number that we've just stored in the ebx register.

Now, the byte PTR and brackets notation tells the CPU treat that number that's in the ebx register as an address. Go there in the bank of RAM and write zero to one bite at that address. The next instruction on in the listing is doing the same operation, except it's writing a full double word into memory at the address pointed to by ebx.

So again, we've already pre loaded EBX with the address we wanna write to. Now the processor sees this instruction with the brackets and the D word PTR notation and says. The address that's stored in the EBX register, I need to go to that address in RAM and right zero to a D word worth of RAM, so 32-bits of zeros.

You can also fetch data from memory by switching the source and destination operands around. So the next instruction moves a byte from memory, pointed to by a constant in this case Foo Into the AL register. So the processor is gonna go into your RAM bank at the address Foo, grab one byte and store that in the AL register.

Now, if you're dealing with more complex data in memory, sometimes it helps to not only have a single base address. But also a base and an index to offset or displace your address into memory. This is very helpful when dealing with arrays for example. So, in the next instruction in the listing, we're actually moving the ad moving the data that's in the EAX register.

Into memory, at the address computed by ECX, plus 8 times EBX. Now, in one step, the processor is gonna do that arithmetic, come up with its final address, and then go there to store the value that's held in EAX. So for example, if this is an array, each element maybe 8 bytes long.

So, the processor can go start at the address ECX and then index into that array, EBX index number of times, and the scale there multiplying it by 8 gives you the final address in memory that you wanna store the EAX value into. Similarly, you can add a constant displacement to the beginning, so you can just skip over the first 100, in this case, bytes.

The last instruction in the listing will take the ECX register as its base, add to that four times in the EBX register. And then add to that, 100 bites, and that will give you the final address and memory that that instruction is going to store EAX into. Now, I know this all sounds complicated right now.

Give it a minute to sync in and we're gonna see it over and over again, and it's gonna become much clear in the future. And finally, let's just touch on how you define data in an assembly language program. So we may have a sequence of bytes defined as friend, in this case, the label, and the value there might be joe.

So you could just list out those bytes individually or as a single string in your assembly code. You could do the same with numbers where you list out say, gross as the label and then give it a data type a D word, and then the actual value. So you could go right ahead and say 144 or 12 times 12, it's all gonna have the same effect.

You can also define arrays, where the array base is the label values. And then each element is a D word and you can give them values by just listing them out with commas. This is gonna give you 432 bit numbers in an array. There's also a way to define DWORD, you can give a label and then the QWORD, data type and then give it a hard coded value.

So more you can give negative integers via the SDWORD or signed the D word datatype. There's also a 10 byte integer called TBYTE. And then the DUP construct is also used quite frequently. If you, for instance wanted to have an array of 50 asterisks, but you didn't want to type out 50 asterisks by hand.

You could use the DUP keyword to just duplicate those 50 asterisks right away. Floating point numbers are defined as a real 4 for a 32-bit flow. A real 8 for a 64 bit flow, and a real 10 for an 80 bit flow. Now, remember that code we saw in the previous slides.

I bet a lot more of this will make sense to you now that you understand some of these instructions and how they work. So in the beginning, you'll see the compare instruction where it's comparing EDI to two. You'll see the jump if equal instruction, that's jumping you to a label.

Further down, you'll see where we're reaching into memory, grabbing a quad-word from RSI plus 8. Those memory fetches look a lot more familiar now. With repetition and looking at assembly code throughout this course, it's all gonna become much clearer. You can check your knowledge right now by just going through this simple example, with what we've already learned.

## L4 Executable Files

>> Hello everyone and welcome back to Advanced Topics in Malware Analysis. In this lecture, we're going to cover different instructions that you're going to see in malware and different executable files that you may have to handle when you're combating malware. So executable files are divided up into sections and these sections are very important for understanding the different pieces and parts of a binary executable that you're trying to analyze.

Sections are denoted by a dot section directive in the disassembly and they can also have some flags or types or even arguments to describe that section. Some of those flags can be a if the section is allocatable or w if the section is writable or x if the section is executable.

You can look these up if you need to when you're reverse engineering, but just being familiar that there there is very important. And then the different types can be, say progbits that denote a section as containing for the program or a note section that is not program data or executable, it's just there as a note to the operating system for example.

If we look back at that code we saw before you can start to notice the sections, for example the beginning of the file begins with a .file section and that's followed by a .section.rodata string 1. So that's a table of strings that are used by this executable. In this case, it's the hello string that we had to say hello to the user.

You're also gonna start to notice the very common Intel instructions that you're seeing over and over again. By the end of this course, these are gonna be burned into your brain exactly how they work. At the top, the most common instruction you're gonna see is the Move instruction.

We've already seen it a bit and you've gotten a feel of how it works. It moves the source operand Into the destination operand that could be registers or memory and you'll see many different uses of this. There's also different set and clear instructions for setting or clearing the flags in the flags register.

And the table on the right here shows which flags are affected and how by each instruction. There's Push and Pop instructions that allow you to push data or onto the stack or pop that data back off of the stack. There's also some instructions for converting bytes to words and otherwise But you've got to be very careful.

These slides are not meant to exhaustively teach you everything about assembly. The Intel manuals and online resources are going be your best friend when you're reverse engineering malware, because there's simply too many corner cases to cover even in an infinite slide deck. For example, if I'm move the contents of the ebx register into the eax register, what is the effect of this instruction gonna be?

At first glance, you might think, it's just gonna move ebx into eax. But what if this is a 64-bit CPU? What is it gonna do with those top 32 bits of the RAX register? It actually automatically zeros out whatever was at the top 32 bits of the RAX register.

That may surprise you if you didn't know that ahead of time and malware authors for sure have looked up this corner case and they're gonna abuse these to trick reverse engineers. The manual actually says, a 64-bit operand, generates a 64-bit result in a destination 64-bit register. 32-bit operands generate a 32-bit result zero-extended to a 64-bit result in the destination register.

And that all sounds good, but how do we handle 16-bit operands? It's actually a totally different story, 8-bit and 16-bit operands generate an 8-bit or 16-bit result in a 64-bit destination register, but the upper 56 or 48 bits respectively are left intact. And this is the importance of reading the Intel manuals and to be sure you understand exactly what the instructions are doing.

Some more common instructions that you'll see. There are arithmetic instructions like Add, Subtract, Divide, and then Signed division or Signed multiplication for cases where you have a two's complement number that you want to add in a signed fashion. Also INC and DEC, which increment or decrement the value in a register.

And then similarly, arithmetic Shifts and Rotate which just move the bits around in the operand register. There are also instructions for making Boolean decisions like negating the value in a register or inverting the value in a register. There's logical AND and OR and XOR operand operators like we've seen in previous slides.

There is a No Op instruction that's simply NOP that tells the CPU to click once and move on. The LEA instruction can be very tricky so keep an eye out for when you see it in an ANA disassembly. It actually computes a destination address in memory and then just stores that address in the destination register.

It doesn't actually fetch the data from memory, it just loads the effective address into the destination. Then there are a number of control flow transfer or Jump instructions. The Call instruction calls a function. The Jump instruction unconditionally jumps to a destination in this case, a label somewhere else in the code.

Then there's ways of conditional jumps like JE jump if equal, or JZ jump if the zero flag is set. You can reference this table if you ever see a conditional jump that you've not seen before to understand exactly what it checks. There's also great resources online to explain these things.

And here's additional conditional jumps that you might see in the wild such as Jump if Above or Jump if Greater.

## L5 Intel and ATT Syntax

>> Hello everyone and welcome back to Advanced Topics in Malware Analysis. Today we're gonna be covering Intel versus AT&T Syntax with assembly language. And we're also gonna go over how the stack and the heap work at a binary level. So Intel versus AT&T Syntax are two different representations of the exact same assembly language instructions.

They just kinda grew up at the same time. One was being pioneered by Intel, one was being pioneered by AT& T. They represent the same thing, but it's like two different directs of the same language. Virtually every tool on Windows uses Intel Syntax and GCC grew up using AT&T Syntax.

So pretty much every tool on Linux is gonna use AT&T Syntax. You can override these two with different switches on different platforms. But as a reverse engineer, you're gonna have to be familiar with both because depending on what tool you need to use, you may just quickly need to be able to understand if you're looking at Intel versus AT&T syntax.

So here's quick cheat sheet of how to tell the differences. Intel instructions are what we've been seeing in the slides so far. So you're familiar with the push instruction, so here we have push four, in AT&T you actually add $1 sign before those operands that are constant integers are constants.

So in this case you would have a push dollar for to indicate that constant. AT&T instructions also add a size denotations to the end of each instruction. So before we were pushing for an Intel, but on AT&T Syntax, it's going to be pushl 4, and l there stands for long or pushing a 32 bit integer on 32 bit machines.

Similar with the add instruction where we used to say add eax 4 that meant to add the number 4 to the value that was in the eax register and then save the result in the eax register. With AT&T syntax, the source and the destination operands are actually flipped.

So you can see we have now add l for long 4 is our source operand. And eax is our destination operand. And you can see AT&T also requires a little percent sign before the names of registers. The size denotation of AT&T syntax actually helps you out when you're referencing memory locations.

So for Intel we use to say move into the al register, the byte pointed to by FOO, in AT&T, that bite size is already encoded into the mov instructions. So you've got mov b, so that tells the processor you're moving a byte, and remember to flip your source and destination.

So mov from FOO into the al register. Call, jump and return also have to have a size even though it's always gonna be a long because that's what you're, that's the size of addresses on 32 bit processors. Referencing memory gets a little tricky when you're switching between the two because AT&T introduced a new way to note memory addresses.

On Intel, we used to have a displacement and then brackets base plus index times scale. So you would start with a base and then multiply your index times the scale, sum those two together and then sum your displacement to that. AT&T has the same components. They're just rearranged.

So now you would have a displacement followed by parentheses, base, index and scale separated by commas. So in Intel where we used to say mov eax bite pointed to by ebp minus 4 in AT&T, you would say -4 (ebp) parenthesis. And you can see how different AT&T syntax versus Intel syntax kind of build up these instructions as they go.

But note, constants in memory references do not need the dollar sign, because the processor knows just by seeing that it's a memory reference, that it is just gonna be a constant. So you don't need to add the dollar sign, you're only adding dollar sign when the constant itself is the operand.

And just to put this in terms that we're all familiar with, I like to say listen to Drake. Intel syntax is much easier to read. So try to stick with that whenever possible. Now we'll talk about the stack. The stack as far as a binary program is concerned is just an endless sequence of memory slots and these slots are allocated in descending order.

EBP remember that special purpose register is gonna point to the most recent stack base address. That's where your function left off. And it's gonna give you a place to kind of reference when you're talking about the stack. The stack base address usually points to where a function call was last made, and it's gonna be updated pretty much every time we make a function call.

The ESP register however, points to the bottom-most used slot. So if you wanna push something new to the stack, you're gonna look at the ESP register for where to go next to place that new value. That brings me to the push instruction. We've seen the push instruction a lot and the push instruction is basically an alias for two separate instructions.

Sub ESP 4 so that's gonna subtract 4 from ESP pushing it down to the next free slot and then mov whatever your operand is into that ESP position that you've just moved ESP down to. So to give you an example, if I push a new value onto this stack, it's actually going to move ESP down, move the value of what I'm pushing on to the stack.

And so you can see that the stack is going to grow downwards. Pop works the same way but in reverse. It's going to take the value of that thing that's on the stack, and it's going to move it into your operand register and then it's going to add for the ESP to move ESP back up.

So if we pop that z that we just put onto the stack off of the stack. It's going to move ESP back up, and we're going to be back to where we started. That's how the stack works. And in general, compilers are going to keep the stack in order very, very rigorously.

So if you're looking at compiled code, you can usually trust that these rules of the stack are going to hold true. Now for how the stack works when you're calling a function, in 16, and 32 bit programs, you call functions by pushing their arguments onto the stack before you make the call.

So you're going to have the function set up by the compiler in such a way that in this case, we have arguments A, B, and C. We're going to push those values onto the stack in reverse order, before we call the footbar instruction, so our arguments are a, b and c.

If we assume that they're stored in the eax, ebx and ecx registers respectively, then we're gonna push those registers in reverse order push ecx, push ebx, and then push eax. Then we're going to make our call to the footbar instruction. The call instruction is actually another alias. Call is gonna push what's called a return address onto the stack.

That means when footbar returns, it's the next instruction that the execution should return to. That's going to get pushed onto the stack. And then control flow is going to unconditionally jump to the entry of the footbar function. The footbar function is then going to push the ebp register onto the Stack.

That's going to save the previous base pointer for the Stack. Then it's gonna move ESP into EPB. That's gonna set the new stack base for where the footbar functions stack frame begins. Then footbar is gonna subtract from ESP, enough room to store all of its local variables. So, in this case x x, y y, z z, and some, each of those are four bytes.

So we're going to subtract 16 from ESP to make room on the stack. This beginning operation for the function where we It pushed ebp, moved esp into ebp and then subtracted ROM for the local variables, is called a function prologue. And you will almost always see this at the beginning of functions because compilers are really strict about keeping the stack in order.

So how do returns work. The return instruction is again an alias. It's actually just pop EIP, it's gonna pop whatever's at the top of the stack into your instruction pointer register. And so that's gonna shift your control flow. Remember, before we call that footbar function, we pushed that return address.

So prior to executing a RET instruction, the function is going to have to store its return value in the EAX register. So if you're say, returning 5, you'll set 5 into the EAX register. And then you have to clean up the stacks so that the thing that's left there is just the return address ready to be popped.

In the case of the footbar function, we ended with this arithmetic. And then we returned to the value that we resulted with, and we returned to the main function and assembly code that looks something like this. That Final Edition will be that add instruction, then the function is going to add 16 back to ESP.

So that's going to clean up those local variables off the stack. It's gonna pop the old base point or the old EBP Value into EBP again. So that when main begins executing at the return address the base point of the stack will be back where it expects it to be.

Then will execute a ret instruction that will pop the return address off of the stack and into EIP immediately shifting the execution back to whoever called footbar. And because we've done the work of popping EBP back into the EBP register. The stack frame will be restored as it was before you called the footbar function.

Things are a little different on 64 bit architectures. Now that you have more registers, we can actually store some of these arguments in registers. On Linux, you use rdi si dx cx r8, and r9 for storing these arguments and you can see It's going to be the first six arguments that you store in the registers.

After that, you go back to the old pushing in reverse order to the stack that we did before. On Windows, things are a little bit different. If you're looking at a Windows binary only four registers are used for passing arguments. Those are cx, dx, r8 and r9. It's just the difference at how Windows versus Linux is designed.

And if we look at this at the binary level if we were calling the foobar function on a 64-bit processor, we would first push the last two arguments because those overflow on to the stack. Then we would just move our additional arguments into the appropriate registers and execute that call.

That would put the return address on the stack. And then everything would execute the same as we saw before. So now let's talk about how the heap works on binary programs. The heap is actually much simpler than the stack, because you have to call a function to get that heap space to use.

The program is going to call malloc, or one of the other keep allocating functions. And then the address that gets returned, is gonna be returned in the EAX register, so you'll just see the program naturally using it. So if we have the footbar function from before, we can see it's just gonna make a call to the malloc function, and that return value is gonna get returned as the address that malloc has allocated and you can use it naturally.

At the binary level this looks just how you would expect. You push the argument to malloc, make a call to malloc and then the return value in EAX gets used. And as far as binary programs are concerned that how the heap works. Here are some great additional reading for you to continue learning more and more about assembly language.

The RE4B PDF that I've put up on Canvas covers Intel, ARM and MIPS assembler with great examples. It doesn't focus on malware, but it's definitely the best way to learn how to reverse engineer assembly language. There's also the Intel architecture manuals that I keep talking about. Please go download those and skim through them whenever you're curious.

The rest of these are online references that you should absolutely check out because each of them provides great background on assembly language and how to reverse engineer. And so that's all for this lesson, we've learned how to recognize x86 assembly, how to read different 16, 32 and 64 bit architecture code.

And we've also looked at executable files, Intel and AT&T syntax and the stack and the heap. I'll see you in the next video.

## L6 Extra-Credit

>> Hello everyone, before we move on to the next slide set, I wanna give you a chance to earn a little extra credit. For this extra credit assignment, we're gotta get your assembly skills up to speed, especially if you've never worked with assembly language before. We're gonna compile that hello world C code that we've been seeing in the slides.

For each line of the assembly, you want to add a comment explaining what each instruction is doing. Be smart about it. I don't want to see any moves to into eax. Of course it does. instead think of what that instruction is doing at a meaningful level in the program.

If 2 is the number of arguments, leave a comment something like the number of arguments must be 2. To submit, take your commented assembly code file and upload it to Canvas. There's gonna be an extra credit number 1 assignment there. This can give you 5 extra points that will fill in last points on real assignments later.

Please check Canvas for the due day. And just for a reminder, here's that C code that we've been seeing in the slides. And this is the GCC command you can use to get GCC to generate an output assembly file for it.