# ECE 6747 Advanced Topics in Malware Analysis

# MODULE 3 TRANSCRIPTS

## L1-Intro to Malware Analysis Tools

>> Hello, everyone, and welcome back to Advanced Topics in Malware Analysis. Today, we're gonna start talking about the static malware analysis tools and techniques that you'll need to know to dissect real malware samples. The learning objectives for this lesson are to learn the different ways to analyze executable files, evaluate different sections at entry points that malware may mess with to make it difficult to reverse engineer them.

We're gonna talk about different static analysis tools of the trade that you can download sometimes or even buy to make analyzing and debugging malware even easier. And we're gonna talk about Ghidra, which is one of the primary tools that we're gonna use in this class for reverse engineering malware.

Static malware analysis is the study of malware code without executing it. This has a lot of advantages. First, you're never executing the malware itself, and that means you're at no danger for executing the payload. You can discover otherwise hidden behaviors that might not show themselves when you execute a malware sample.

It may be doing some checks that make it very difficult to exhibit some of these behaviors. But if you're studying the code as it exists in the file, you can often discover a lot of these hidden behaviors. But there are some disadvantages to static analysis, the primary one being encryption and packing.

We're gonna talk about both of those in depth later in this course, but to begin with we're not gonna worry about it when we learn about static analysis tools. Another problem with static analysis is complexity. You're gonna see this in your upcoming assignments, because statically trying to understand what code does at the assembly level can be extremely challenging.

Great tools like Ghidra are gonna try to help you, especially by trying to help you understand high level language structures, which can be very complex to look at in binary code. But in general, static analysis is one of the primary tools that reverse engineers will go to when trying to understand malware.

There's a few tools of the trade that you're gonna need to know to do this analysis. EXE manipulation tools, that is executable file manipulation tools, such as DUMPBIN, PEview, PE Explorer, allow you to analyse the structure of an executable file. You can dump code sections, data sections, the import table, the export table.

We're gonna go over all of these later on in the lesson. Disassemblers allow you to analyze the code. This is like Ghidra or IDA Pro. But these allow you to actually look at a disassembly of the program and try to understand what the code is doing. Usually you can add some comments or even run automated plugins to help you do this analysis on the code.

Then finally, there are decompilers. Decompilers attempt to translate binary code Into a higher level language, and you're gonna see this when you start using Ghidra. Analyzing executable files can be quite challenging. Remember, they're divided up into sections. There's also entry points and dependencies that you'll see and have to understand when you're reverse engineering malware.

There's a symbol table, which may or may not be present if the code is stripped, code, data, and other relocation information. Tools like DUMPBIN on Windows, PEView on Windows, readelf for Linux and otool on Mac can allow you to look at these different features in an executable file so that you can figure out how the malware has manipulated that file in order to run.

However, these tools are not a replacement for your deep understanding of the executable formats. These tools are simply going to show you these different portions of an executable file in a human readable way. You have to understand what the tool is trying to tell you. This is gonna be quite challenging and you're gonna have to read, study, Google, and learn everything you can about executable file formats when you're analyzing malware.

Now, I know what you're thinking. Must I? Yes, yes, you must, especially if you're dealing with malware. Old school malware, like running on Windows previous versions or DOS even used to just get away with being a regular executable file. But nearly all modern malware heavily abused the executable file formats to infect and propagate.

When you're reverse engineering malware, code is gonna be very opaque to you unless you understand the executable file formats. Reverse engineering is often about these nasty details. I'm sorry, but that's just the name of the game. You're gonna be on Google a lot. You're gonna see a malware reach out to a strange offset in some header in the file, and it's not gonna be clear what that header is trying to do until you've understood the entire file format.

## L2- Executable Formats – Elf, Windows PE, Optional Headers

>> Hello, everyone and welcome back to Advanced Topics in Malware Analysis. In this lesson, we're gonna talk all about executable file formats. Executable file formats allow the loader to instantiate a new program at execution time. The loader is just another program or a part of the operating system that reads that executable file off of disk and starts the process running.

Compilers may even insert some setup code before your main function gets called. This helps handle the transition from the loader to your program. It does things like initialize the data or startup any libraries that you're trying to call. You can read more about it in the documentation for the tmainCRTStartup function.

That's one of the most common startup functions that gets inserted before your main function on Windows executables. Programs then execute machine code directly on the processor and interface with the runtime system just via regular library calls. Executable file formats have evolved over time. Back in the days of DOS or very early Windows, executables evolved from what was called COM files.

COM files were restricted to only 64KB in size and you can see how that limit quickly got to be insufficient. EXE files were introduced in 16-bit and 32-bit mode WIndows, and have now evolved into 64-bit Windows. On Unix, ELF files or executable and linkable format is the most common by far and you'll find ELF files on anything that's based on Unix, Linux and even android operating systems.

The ELF format on Linux allows for two different interpretations of the file. You can look at a file via segments which contains the permissions and the mapped regions of the file or by sections which enable linking, and relocation of the file. The loader, when you start up that executable running is first gonna read the ELF header from the file on disk.

Map those segments into memory in a new virtual address space. Resolve any relocations like external library calls and then start executing from the entry point. You could also have an .interp section. That's optional and that tells the loader, and interpreter that you wish to use to help bring your program to life.

There are a few tools for inspecting ELF executables. Readelf and objdump are by far the most common. They can show you things like the different headers and the different segments within an executable file for any ELF file on a Linux system. They also work on executables, shared objects, archives and even object files.

But your brain is much better than those tools. Many different ELF specifications exist for different platforms. Wikipedia has a great reference if you follow that link. But you, yourself have to be able to read and understand the output produced by readelf. For example, readelf -h is gonna give you all the information about the ELF files header.

-l is gonna display all of the headers used by the loader and -S is gonna show you all the sections. On Windows, you have a different file format. It's called the PE or portable executable format and this is actually used for any executable file on Windows. EXE, SCR files and even DLLs.

There's no distinction between these. The only difference between an EXE and a DLL is that an EXE has an entry point, and DLLs have exported functions. The file format consists of a number of sections including sections for backwards compatibility with MS DOS. Interestingly enough, if you run even a modern windows executable on an old DOS system, it'll still run.

But it'll print a warning saying, I can't execute this program on MS DOS. The Windows PE format is a little different. There's a complete specification of the PE format available from Microsoft. You can Google it and download the entire thing for yourself, which is very helpful from a company that tends to keep things closed source.

There's also an outstanding blog post by Microsoft engineers condensing that entire format down into the key things that reverse engineers have to understand. It's very important that you be familiar with all of these details of the PE format. Because I assure you, malware authors definitely are and they're gonna abuse all of these features in order to make it difficult for you to reverse engineer their malware.

One key thing to understand about PE files is the difference between alignments when the file is on disk versus when the file is in memory. Typically, files are aligned to 512-byte alignments. That means when the file is on disk to try to save disk space, all of the different sections and segments will be aligned to 512-byte divisions.

When allocated into memory, memory pages are allocated at 4KB of size. And so those alignment boundaries get blown up when that PE file starts executing in memory. This leads to a difference between the addresses on disk versus addresses in memory. Windows calls these relative virtual addresses. Basically, Windows normalizes all of the addresses within a section to the base address once that executable is loaded into memory.

You can read more about these relative virtual addresses on Wikipedia, but it will be very obvious when you see them, because they're going to be offsets from the load address of the executable. The COFF header is a header in the PE file that describes a number of important flags for how that executable should be loaded and run.

I'm not gonna go through them all, but you can use this slide as reference when you need to look any of them up. There's also an optional header. Now it's called optional, but it really is only optional for object files. All PE executables are gonna have this optional header filled in.

This contains additional fields that describe Windows specific operations that allows the program to execute. Most importantly, it contains the data directories and this is where very important fields for the program's execution will reside that malware might try to mess with. Also in the optional header, things like linker versions, sizes of the code and sizes of the debt data sections.

Again, malware will easily reach in here and increment these if they're trying to add their code to other executables. A very typical virus operation. There's also base addresses stored in the header in the optional header that you'll need to be aware of and most importantly is the image base.

Remember those relative virtual addresses? The image base is where those addresses are computed from offsets of it. So you wanna look for where that image base is and then all of the relative virtual addresses will make sense to you, cuz you can just add those offsets to the image base.

More windows specific fields like the operating system version and the image version, all of these are gonna be important, especially the size of the image. Cuz if malware tries to append their code to another executable, they're gonna need to account for each of these fields. Missing one could tip off reverse engineers like yourself to the presence of that malware and then additional ones like size of the headers or even check sums that need to be accounted for.

And finally, the required heap allocations or the number of relative virtual addresses that you would find in the data section. The data directories that we were talking about before are actually a data structure containing an array of these image data directory structures. Each one of these is a DWORD for a VirtualAddress and a DWORD for the Size.

Together, an array of these forms a directory which allows you to interpret the different sections in a PE file. In the next slides, we're gonna see how that all works.

## L3- Data Directories, Section Headers

>> Hello everyone and welcome back to advanced topics in malware analysis. In this video, we're gonna be talking more about data directories and section headers, and different tools you can use to inspect them. So recall from the previous slides that each one of these data directories is an array of those data directory data structures.

In terms of what data directories a PE file has, you're gonna be looking for the export table, the import table and the resources table in particular when reverse engineering. Certainly, these other data directories matter, but malware are really gonna try to hide what functions they're exporting or importing.

Similarly, you wanna look for the IAT, the import address table. This also describes the functions that a PE file wants to import from shared libraries. Also improtant for reverse engineering are the section headers of a PE file. Here are a few that are particularly important for reverse engineers.

And specifically, the global pointers that help you locate relative virtual addresses. The section headers area follows the optional header in the PE file. It contains a series of 40 byte entries, each entry describes one of the sections in the file. The section headers are followed by the section data, such as the .text section, the .bss section, etc.

These section headers are things like the name of the section, the virtual size of the section, and the virtual address at the section is allocated to. Similarly, raw data and pointer to the raw data, and then many other section headers that are gonna be important when you reverse engineer.

When you reverse engineering a PE file, you wanna keep your eye out for these important sections. The bss section stores uninitialize data, the data section stores initialized data. The edata and idata sections hold the export table and the import table. And the .text section stores all the code for that executable.

There are others, these are the most common and this is also the most common naming convention for them. You can see different names, but it's gonna to be strange to see them. And when you see names that are outside of this common section names, it might tip you off to the presence of malware.

Many tools exist for doing static analysis of PE file sections. My personal favorites, Dumpbin, PEView, and PE Explorer, but feel free to get your feet wet with any tools that you can find. I'll give you an example, here is the Dumpbin program run on notepad.exe. You can see that this shows all the different file headers in that PE file, and their values.

Pay close attention to the timestamps, the file pointer, and especially the optional headers. The optional headers have that magic number that says that this is a PE file. It also has different versions of the linker. And remember those relative virtual addresses we talked about. Here is the entry point, which is a relative virtual address itself and the image base.

Now if you add the entry point to the image base, you get the true entry point of this executable. That's where the execution is gonna begin when the loader finishes loading notepad.exe. If we keep scrolling down in Dumpbin results, we can see it shows all of those data directories, the export directory, the import directory, all those directories that we talked about before.

Dumpbin.dumpbin then walks through notepad.exe's sections, and displays all of the section header information. We can see the virtual size of the .text section. We can also see the flags for the .text section. It's no surprise that that's marked as code executable and readable. If we keep scrolling, we can see the additional sections that are in notepad.exe.

There's that .data section that we talked about before, also the .resource section and .relocation section. Now notice, any pointers that are in the section headers are in terms of relative virtual addresses. And you can add the base address to those to understand where they're gonna lay when they're put in memory.

We can also use Dumpbin to list the imports for notepad.exe. Here we can see notepad.exe is importing quite a few functions from shared libraries, such as Page Setup from comdlg32.dll. And if we look down, we can see Winspool driver library as well, and that makes sense. Of course, notepad.exe is gonna need to print files.

This is another tool that I really like called PEView. If we run PEView on notepad.exe, we get the same file and section headers as before, but now they're in this nice collapsible list that you can scroll through. And it even gives you a helpful hex and raw value view of that PE file as you're inspecting the different headers.

Believe it or not, the strings command can even be helpful for reverse engineering executable files. For instance, this was how the earliest signatures of malware were made. Running strings on a malware is gonna print all of the ASCII printable strings that are contained in that file. You can do the same for Unicode with different flags.

It may not always be helpful, especially if the malware is packed or encrypted. But memory forensics has even used strings extensively to try to find indicators that a malware is present in an executable file. Here's an example, this is the Gruel Worm from Windows XP days. When this malware was going around, you could easily detect it by running strings on its binary.

The nonsense email that the spam worm was putting out would spit out immediately from the strings of the executable. Here we can see the craziness that the malware author had written. Similarly, a Unicode version, two different versions of the same worm, both detected by the simple strings command.

In this case, the malware author included a fake Norton Antivirus response in the email. But you can notice a few typos, so that might have given them away. And that concludes our discussion of static analysis tools for PE sections.

## L4- Packers

>> In the last lesson, we learned all about understanding executable files. And that works out well, until you deal with packed malware. Packers are gonna do everything they can to obfuscate all the lessons that we just learned. So in this lesson, we're gonna learn how to counter those packers when you see them in the wild.

Nearly all state-of-the-art malware does not want to present an easy target for you to reverse engineer. What we've learned so far has been understanding a reasonable disassembly. Once you can actually see the sections of an executable file and the assembly instructions that you're looking at. But modern malware is gonna do everything it can to prevent you from even accessing that clean disassembly.

Further, modern malware definitely don't want you to be able to analyze its behavior at runtime. So packers have evolved over time with new generations of malware to add more and more layers of difficulty to reverse engineering. The basics of a packer are that all packers are gonna add obfuscation/encryption routines to parts of the viruses body, in order to make your analysis more difficult.

Packers are gonna hide one binary inside of another binary, often called the host binary. And then only reveal that hidden binary when it needs to possibly decrypting only one piece at a time, when it goes to execute those payloads. This can make reverse engineering very difficult and there's a few tricks that you're gonna need to learn in order to counter these packing attempts.

So the basics of packers. Packers have the following general structure. They're in a compress/encrypt a target executable files code, and generate a new executable with the encrypted code as a payload inside of this new benign looking executable. The new executable is gonna have a modified entry point, that's gonna point to some unpacking routine.

The unpacking routine is responsible for decrypting that packed code that's hidden from you. And then it may even contain some anti-debugging tricks to try to trip up any runtime analysis that you're trying to do. The unpacker's ultimate goal is just to decrypt and jump to the original entry point of the packed code.

Now, packers are not just an outright malicious thing, packers are actually used on commercial software all the time. A good example is a lot of Microsoft products Word or Skype, are gonna come packed. This is not to be malicious, this is to prevent people from stealing any trade secrets that the code has.

So you can't just assume any packed binary is malicious, but it's a very common practice for malware to stop you from reverse engineering it. So when you're doing static analysis, the first question you're gonna wanna ask is, is this malware packed? And if it is packed, you're gonna have to determine what type of packer you're dealing with.

Is it something that's well known? Is it a hand-rolled packer that the malware author wrote themselves? Luckily for you, there's some tools that can help you with this. This is a favorite of mine called PEiD. PEiD comes with signatures of known packers, that you can upload a malware to and it's going to identify what packer was used to make that packed executable that you're looking at.

PEiD is been a fan favorite for a while and there's many others out there. But PEiD comes with a big library of over 600 different types of packers that can identify. In this case that you're seeing on the screen, I've packed a little HelloWorld executable with the Armadillo packer that we'll talk about later.

Armadillo is an outstanding packer that's used both by malware and regular commercial software. PEiD is gonna go a step further and even try to show you what section of the executable contains the packed data. And that can help you for starting to tease out some of that malicious functionality.

But you may be dealing with something a lot more insidious. You could have an entirely home-made packer, in which case PEiD or similar tools are not gonna be able to help you. In some worst cases, you could be dealing with even an XOR-based stream cipher, where the key is hidden somewhere in the virus body and you've gotta pull that key out.

Run the XOR yourself in order to get the decrypted virus out to you. There could be a pseudo random number generator somewhere embedded in the virus body. That you have to reverse engineer and figure out how to run to get those keys generated for you when you need them.

Luckily, the key has to be stored somewhere in the virus body. So that allows you to do this on packing although, it may be a little difficult. Unless you run into the real worst case, and this is a true situation that we've run into in my lab. The key to unpack the malware could be downloaded at runtime from a command & control server.

So when you're looking at the binary itself, you're not gonna see a key anywhere to unlock those hidden parts of the malware. But things get even worse. In the research, we've even seen cases of malware that have no key at all. The malware actually decrypts itself over and over again, and checks a hash of the result against a known hash of the virus body.

So you would have to literally compute a hash collision to get that malware to reveal itself. If you're more interested in this, there's a great PDF that you should read that I've put up on Canvas. It's actually a publication from Blackhat a few years ago, called The Art of Unpacking.

That looks at a bunch of different unpacking techniques that reverse engineers can use to get malware to reveal itself. If you wanna read even more, there's an outstanding research paper published by a researcher named Kevin Roundy, called Binary-code obfuscation in prevalent packing tools. This paper goes over a number of tactics that you'll see in modern malware to try to keep their code packed and make it difficult for you to reverse engineer.

If PEiD or other tools fail, you often will have to resort to manual inspection. And next, we're gonna talk about the different ways you can inspect by hand to see if a malware is packed and what suspicious things you should be looking for. So let's start with the section names.

Remember the sections that we talked about before? A packer is gonna try to hide the packed code somewhere in those sections. So you already know the typical section names that you should be looking for, keep your eye out for any unusual or suspicious section names that you've never seen before.

Here's an example. If we look at the section names from the unpacked HelloWorld executable, we see the data section, the read-only data section and the text section. Nothing unusual there. But once I've packed that executable with Armadillo, we can see new sections there that really stand out. We've got this strange adata section, this data1 section and text1 section that we've never seen on any other executables.

This is definitely suspicious and it should flag you to start looking at these in more depth. Another thing you should inspect are the permissions of each section. What's marked as executable? What's marked as code? What's marked as data? Here we see that .adata section from the packed HelloWorld executable.

It seems awfully strange that this section is marked executable, read and write. Why would you need all those permissions on the same section? Definitely clues you in to dive a little deeper into that .adata section. Here's another example. On the benign HelloWorld executable, we can see that the entry point, points to the .text section.

This makes sense. The entry point of the executable would be in the regular .text code section. However, when we packed that executable we can see that the entry point now points into a different section, no longer into the .text section. That's pretty unusual wouldn't you say? If we dig into it a little further, do we see that the entry point actually points into that strange adata section?

So that definitely clues you in to dive deeper into that adata section. There's clearly something going on there that's unusual from a regular executable. And we'll get back to more executable file formats in the next slide set.

## L5- Finding Strings, Tables, and Code

>> Hello, everyone, and welcome back to Advanced Topics in Malware Analysis. Today, we're gonna continue talking about how to find packers by looking at the different strings, and tables, and code sections you can find in the executable file. Manual inspection of the executable can often reveal printable strings, that can give away if an executable is packed or not.

And you're probably thinking, strings again, this is pretty low tech, but that's true. But what you can find by looking at the strings is if common strings that you would expect to find in any executable file don't show up. And if they don't show up, that's a good indicator that the executable is packed.

So if you run strings against the executable file, look at what you see. If you're missing things you would expect to see from an executable file, for example, if you know it connects to the Internet, but you don't see any URLs, that's probably a dead giveaway that that executable has been packed.

Another great place to look for packing is in the imports table. Why does the imports table look suspiciously empty? Maybe because a packer has hidden from you all of the different functions that that executable is importing. Most windows programs have a huge number of imports of functions that they're trying to call from different libraries.

Here is just the strings executable that I've run a tool that I've run dump bin on in order to get the imports table. You can see from just this one DLL, kernel32.dll, this executable has imported a huge number of functions that it's gonna need when it's running. However, after we pack that same executable, you don't see that many functions being imported.

And the ones that you do see imported are pretty suspicious. Things like load library, get proc address, other functions that would allow you to look up the addresses of new functions at runtime. That's definitely an indicator that something suspicious is going on, and you could probably assume that it's packing.

Now this won't be found just by straight static analysis, because the additional functions that this binary needs are being looked up at runtime. So you're gonna have to dig into the binary a little bit further, to figure out what functions it's actually importing at the time that it needs them.

And finally, if all else fails, you can fall back to manual inspection of the code with a disassembler like IDA. Do large parts of the executable, specifically targets of JMPs or CALL instructions seem to be obfuscated? That's a good way to start suspecting that your executables been packed.

Ultimately, as a reverse engineer, your goal is not to get bogged down by all the packing schemes that are out there. You want to figure out what the malware is doing. So as quickly as possible, you want to isolate the packer code or the decryption routine. And get it to spill out as many of the malware's payloads as you can.

Essentially, you're trying to find the point where the decryption is complete, and you can start looking at the Malware's actual code. How do you do that? Well, in a real case study such as the Lucius Malware, for example, where that malware uses a hand-rolled encryption loop for the virus body.

You have to keep walking through the code slowly but surely, until you see all of the virus code spill out. Once you get the code to spill out, you can start to identify the entry point of the new payload. This is back to static analysis. You just have to look at it.

And I know, you're probably starting to get exhausted already, because this is gonna be a lot of time spent in a disassembler like IDA or Ghidra. Single stepping in a debugger or an emulator can also help. Using a tool like OllyDbg which we'll talk about later in the course, allows you to get a much deeper understanding of the effects of the unpacking routine without having to worry about digging into the instructions.

Because you can basically single step through the unpacking routine, until you see new code getting spilled out. The disadvantage here is that number one, it's slow. Because you've gotta keep your brain ahead of the debugger. It's also very dangerous, because it's very difficult to figure out where the unpacking ends and where the kill everyone payload begins.

So you've gotta be sure to keep yourself ahead of the debugger. You can also scan the function call graph in IDA Pro or Ghidra. A packed executable is not gonna have a very deep function call graph, because all of the functions that you can see are the ones that do the unpacking.

Once the code has been unpacked, you should see a much deeper function call graph that you can inspect for different payloads. One disadvantage here is that tightly rolled packers, such as the ones that only reveal small chunks of the binary at a time, are still not gonna reveal very much when you're looking at the call graph.

Another way that we use to get around packing in the real world is to use memory forensics. You can actually run the binary in some kind of safe or contained virtual machine, like we'll talk about later in the class. And then once the binary has been unpacked in memory, you can dump the newly unpacked code into a brand new executable file on disk.

You can then reconstruct the imports table based on the execution that you see. And in most cases, the unpacking will be complete by the time this code starts running in memory. You can then use your new dumped unpacked executable to begin static analysis. And now I'll give you a case study of a real packing solution that we've dealt with on a number of malware in my lab.

This is called Themida. And it's a commercial packer produced by a company called Oreans Technologies. Typically, this is used to protect commercial software. It uses encryption and a lot of different advanced anti-debugging tricks. And we've seen it used on malware for just this reason. It also has anti-acquisition tricks that prevent you from dumping the process' memory.

This can make it very difficult to combat Themida packed executables with memory forensics, like we talked about before. They also do tricks to make your reverse engineering more difficult, like injecting garbage instructions into the pipeline of the executable. Continuing to talk about Themida, there's also a kernel mode component that breaks any attempts to dump the process from the kernel.

And there's VM-based emulation of x86 code to give it an added layer of protection, if you want to try to inspect the executable instructions themselves. We'll talk about this later on in the class as well. Themida is definitely used by a number of malware variants, because it's so good at defeating anti viruses.

Themida pack executables have become so difficult to reverse engineer, that many secure environments just simply disallow Themida packed executables entirely. And just some icing on the cake, Themida actually has a beautiful user interface when you're trying to use it. You can just spin up your beautiful instance of Themida, give it an executable that you wanna pack.

And then check the boxes of the different options that you want to use when packing your executable. This is what's made it a favorite among commercial tools that just want to pack themselves to protect trade secrets. Another case study we can talk about is upx, upx is actually an open source packer that's widely used by both malware and commercial tools.

One great trick of using upx is you can use it to practice reverse engineering skills, because upx gives you a -d flag that you can run with to automatically unpack any executable for you. This will allow you to get your hands wet, reverse engineering a packed executable, and you can always get the ground truth whenever you need.

In the next lesson, we're gonna start talking about more file formats and how you can use them to combat packers.

## L6- Static Analysis Tools and Techniques

>> Hello everyone, and welcome back to Advanced Topics and Malware Analysis. Today we're gonna be talking about static analysis tools and techniques that you can use to combat those packers that we talked about before. The Swiss Army Knife of static analysis is definitely the disassembler. Disassemblers automatically process an executable and generate an assembly source code for that executable.

The difficulty here is that code written in a high level language such as C or C++, when it gets compiled into a binary is very lossy. You lose all the context of objects and data structures. And you get all the code and data mixed up, so that your job becomes very difficult just trying to figure out what is code, and what is data, and pulling those two things apart.

We'll talk about the different approaches to tackling this problem. But also computing branch targets that are dynamic, or computed at runtime can slow down your reverse engineering when dealing in a disassembler. Dynamically loaded code like DLLs might not be available to you when you're looking at a program statically in a disassembler.

And branches that don't target the beginning of an instruction. Believe it or not, that happens all the time in malware, such as jumping inside a multi byte instruction, and starting to interpret those bytes as entirely new instructions. That's all gonna trip you up in a disassembler, and you're gonna wanna use the best tools that you can to make this easy on yourself.

A few other things to watch out for, self modifying code, or deliberate attempts to obfuscate PAC or encrypt the code that you're looking at, such as the armadillo Packer we talked about before. Luckily for you, some tools can come to the rescue. IDA Pro, for example, allows you to emulate certain instructions in a given piece of code.

So if you start to reverse engineer a malware, and you think that you've found a decryption loop that you want to run through, you can actually tell IDA Pro to start executing the instructions at a given point. And run through until a break point later on. In this case, we could set a break point at the jump instruction at the end of the decryption loop, step through the instructions, and see the new decrypted code that gets spilled out.

IDA also analyzes the file format that we talked about before. IDA can present to you the different sections, such as the dot txt, dot data, or even the C string section, and all of the different contents of those sections. So for the text section, you'll see code, the AI data section, you'll see the imports, and the string section you'll see the table of strings that's in that executable.

Ghidra also works great for analyzing the file format. It stores each of the different section headers in the Program Trees window. Double click on each one of those, takes you to the relevant section in the code. You can then add labels or comments to the code directly in Ghidra.

Another thing that makes reverse engineering binaries difficult, is that high level language to binary code compilation is very lossy. We'll talk about this in a whole lesson later on in this course. But for now, understand that it's a one to many translation, when you compile high level language code.

The compilation process is not unique, and it can choose many different implementations of high level code, when it's putting that into a binary. Data types are gonna be lost, as well as any names or useful symbols. That's gonna make reverse engineering and debugging code, especially from malware that has been stripped very difficult, unless you have the right tools for the job.

The intention of a programmer, like a malware author is then to further obfuscate the code that you're looking at. So you're gonna need to know the different tools you're working with to combat these trends. Disassemblers like the ones we've talked about, actually come in two different flavors. There's Recursive-descent Disassembly, that attempts to reconstruct the assembly instructions of a program by following the control flow.

It's going to disassemble sequences of bytes only if they can be reached from another valid instruction. This is good, because it allows you to handle interleaved code and data better. If you have some data in the middle of code, you won't ever try to disassemble it, because you'll jump over it when you're following the control flow.

It's bad, however, because it can't handle indirect jumps or self-modifying code very well. Because if the disassembler can't figure out the control flow, it can't disassemble those instructions for you. Examples of these are IDA Pro and GHIDRA and OllyDbg, which is a debugger that we'll cover later in this class.

The other flavor of disassembly is Linear-sweep Disassembly. This starts from the very first offset in the binary, and disassembles the entire binary as a stream of bytes. The next instruction is assumed to be whatever follows the previous valid instruction. This can be bad, especially for dense instruction sets like Intel, because it's not easy to tell when you're off track.

Instructions can be one byte, two bytes, three bytes as many bytes as they need to be. And so you get off, by just one byte somewhere in your stream, it's gonna corrupt all the instructions you're trying to disassemble. It's also easily tripped up by interleaved code and data for the same reason.

However, linear-sweep disassembly is good when you're going for coverage. Because indirect jumps or calls don't trip up a linear sweep because it's just gonna plow through all of those bytes as if they were instructions. WinDbg, SoftICE, gdb and objdump are all linear-sweep disassemblers. Then there's been plenty of research trying to combine these two, into different hybrid approaches like first trying recursive-descent, and then linear-sweep and noting the similarities or differences.

There's also an interesting idea of speculative disassembly, and that is, mark portions of the binary that you have disassembled and you're confident in. And then speculatively disassemble other portions and ask, what happens, is this does this look like code? Maybe there's a human that you can ask to intervene and mark known good portions of disassembled code.

A great description of a hybrid disassembly solution was published at reverse engineering workshops a while ago, and you can read that paper linked below. And in the next slide set, we're gonna continue to talk about different tools and techniques for combating static analysis problems.

## L7- Decompilers

>> Hello everyone and welcome back to advanced topics in malware analysis. Today we're gonna talk about the king of static analysis tools, decompilers. Decompilers have been a goal of the reverse engineering community for a long time now. And ideally what you want a decompiler to do is to take the disassembly that was constructed by a disassembler and reverse the compilation process, bringing you back to some kind of high level language that reveals more of the original source than you would get from just the assembly language.

Now, producing the original source code itself is impossible because of how lossy the compilation process is. But you can get close by revealing things like if statements, switch statements and loops that make reverse engineering much easier. There are some very limited open source solutions such as RetDec, that allow you to do decompilation on pretty much arbitrary binary programs.

Commercial systems that do decompilation are very good but have been known to be quite expensive. IDA Pro, the company, Hex-Rays, that makes IDA Pro actually sells a Hex-Rays decompiler that's probably the most famous and does a fairly good job. However, it's very, very expensive on the order of thousands of dollars per year, so only the better research groups at Georgia Tech have access to this extremely expensive software.

I can give you an example of what this decompilation looks like, using this simple string copying C program. This is the original source code, and if we compile that into a binary, we can then give the binary to a disassembler, like IDA Pro. Here you can see the disassembly for a portion of that code.

This makes sense given all of the disassemblies that we've seen so far in the class, but it still requires a lot of work for you to understand what each of these instructions is doing. And since most of these constructs are just if statements, branching and calling functions IDA Pro's built in decompiler, called the Hex-Rays decompiler, can actually lift you on level higher than this and produce conditionals and function calls as you would expect to see in a regular C program.

Here's an example of that. You can see that this isn't the original source code that we were looking at before. And it still requires combing through by hand by a skilled reverse engineer. But it does make your life a little bit easier, allowing you to get through some of the more arduous tasks of reverse engineering with a little bit more simplicity.

Things changed however, when Ghidra was recently released. Ghidra comes out of the box with a built in decompiler. And just like the whole Ghidra platform, it's completely free for anyone to use. This is one of the primary reasons that Ghidra has been so widely adopted so quickly. Ghidra's decompiler synchronizes at the same time with the code view while you're looking at the disassembly.

You can look at the same instructions and decompiled code right side by side. And when you highlight different names, or add comments in one window, they show up in the other. You'll see this when you're using Ghidra in your homework assignments, feel free to go back and forth between the decompiler and the disassembler view.

You're gonna find that there's some things that are better to comment in the decompiler and some things that are easier to understand in raw assembly language. That'll make more sense as we reverse engineer more and more malware samples in this course. And you see the corner cases that a decompiler can handle.

There's gonna be many cases where Ghidra just throws his hands up and tells you it can't decompile a certain part of code because it's too complex or it's packed or self modifying. Of course, since Ghidra is open source, there's already been some wily hackers that have taken its decompiler and made a plugin for IDA Pro, allowing you to use Ghidra's free decompiler in IDA Pro and avoid paying IDA all those giant fees to use its decompiler.

I've put a link down at the bottom if you want to take a look at that. And that brings us to the end of our lesson here. Just as a summary, we've learned how to analyse executable files and reveal the different sections, entry points and dependencies of an executable file.

We figured out how to use those to identify packers and even combat some common packing routines. We figured out the different static executable inspection tools that you can use to do this analysis. And even a little bit of debugging, which we'll get into later in this course. And we've talked about how Ghidra and IDA Pro reverse engineer disassemble, and even decompile executable files for you.

I'll see you in the next lesson.