# ECE 6747 Advanced Topics in Malware Analysis

# Module 7: Dynamic Malware Analysis Tools and Techniques TRANSCRIPTS

## L1. Advantages and Disadvantages of Dynamic Malware Analysis

>> Hello everyone and welcome back to Advanced Topics in Malware Analysis. Today, we're gonna get into Dynamic Analysis. In this lesson, we're gonna try to cover different techniques for performing dynamic analysis of malware. The tools and techniques you're gonna need to do this in the world. And the basic techniques you gonna need to sandbox malware and inspect the execution at runtime.

So welcome to Dynamic Analysis. Dynamic program analysis solves problems by inspecting the execution of software. This is very different than inspecting static binaries. Specifically, not all of the statements that you see when you look at a binary statically, will actually be executed. Instead, you're looking at statements one by one, as they execute along a single execution path.

Even better, all of the variables have been instantiated. There's no more aliasing, which we all know can be quite a challenge to overcome when doing static analysis. Dynamic analysis, removes the problem of aliasing, because you're actually watching a program execute. And so all memory addresses are concrete. You can simply fetch data out of memory, when you need to know what's there.

This results in dynamic analysis having a relatively lower learning curve, than static analysis. We'll be able to get into it much quicker. And you'll find that the results you get, tend to have much better precision. You can also perform much more widely applicable analyses. Because you can simply watch the side effects of a malware executing.

There's also pretty good scalability doing dynamic analysis, because you don't have to worry about that path explosion problem that we face, in the previous labs. Now, there is only a single path of execution to worry about. And everything that's going on along the single sequence of instructions that`s executing.

But be aware, programs in memory, look very different than binary files on desk. It's the job of the loader to map all of the runtime sections, from a binary and any shared objects that it needs into a virtual address space. The loader, then calls global initialization functions to get all of these libraries set up properly.

Can you think about why the order of calling these global initialization functions is important? For example, lib C is gonna initialize heap data structures. So if these are not initialized first, before other libraries are loaded, attempts to allocate memory on the heap are gonna be a huge failure.

So in summary, a lot of what you've learned from doing static analysis is still gonna apply here. But you need to always be aware that executing binaries, are not static files. Things are changing with every instruction that executes, even more interesting, it's possible to map back from a running process to an executable file on disk.

And you can actually dump our running process into an executable file to be inspected with Ida pro or Ghidorah. We'll see how to do this later on in the lessons. And finally, talking about dynamic analysis of malware can be a little tricky. You need to be careful, because you are actually executing a live malware sample.

This is some pretty big advantages. Specifically, you get much better insight into the behavior of that malware, without having to do a deep dive into the code that you're looking at. This also can help with encrypted or packed malware samples, because just running the malware, will naturally get the malware to unpack itself for you.

The obvious disadvantage of dynamic analysis on malware, is that you can and often will, accidentally execute dangerous malware payloads. This is no joke. And in order to prevent this from destroying your box, you need to be very careful, to run malware only in very highly engineered sandbox environments.

We'll talk about these later in the class. When you're doing dynamic analysis, you may also have to overcome several anti-analysis behaviors. Just like packing prevented you from doing static analysis, malware authors have a number of tricks to prevent you from doing dynamic analysis. There are some tools that can help you.

But you always need to be aware that malware know about these tools too, and we'll be actively trying to make it more difficult for you to reverse engineer them.

## L2. Basic Tools

>> Welcome back, everyone. Today we're gonna be covering the basic tools of the trade for doing dynamic analysis on malware. In this lesson, we're gonna cover all of the different basic dynamic analysis tools and techniques you can use to watch the side effects on a system that a malware is causing.

Things like changes to the Windows registry or new files being dropped, can give you a great indicator of how the malware is affecting the system that it's trying to infect. Then we're gonna get into more fine grained tools like system called traces, library traces and process monitoring. Dtruss is a trusted favorite of reverse engineers for tracing.

System calls made by a program. If I zoom in on this large trace produced by dtrust, we can see that it actually shows all of the system calls that this process made, in the order that they were made, along with the arguments that were given to that system call and the return value that that system call returned.

By piecing these together, you can get a great idea of what a malware was doing in a system. Now, you might not want to go through this by hand because it's going to be quite large as you can see from the zoomed out photo on this slide. But if you write some scripts that can parse this output, you can pretty quickly match up known signatures of malware behavior with what system calls are being executed.

Researchers, such as the PhDs in my lab, will often save these to a file and perform offline analysis for thousands of malware, looking for commonalities between the behaviors that they see in different malware samples. Process monitoring is an extremely powerful technique for watching how a system is affected by malware.

Hands down, the best tool for this on a Windows system, is the Procmon utility. Procmon is actually free and comes as part of the windows sysinternals tool suite. Sysinternals contains a number of really excellent techniques and tools for monitoring the changes in a Windows system. But Procmon is definitely my favorite.

It allows you to track all kinds of events that go on in the Windows system across every process that's executing. You can then use very fine grained filters to narrow that list down to only the processes that you're interested in or the files that you're watching. And you can get extremely granular results on exactly what happened on a Windows system while all the processes are still running.

More basic dynamic analysis tools. Network traces and network captures can allow you to analyze the data that was sent or received over the network by a program that's running on your system. We're gonna talk about WireShark in specific, because this is probably the most powerful example of a network trace and analysis tool.

The most powerful tools even allow you to reverse engineer specific network protocols that are used by an executing an application. A lot of research has been done in this field. If you want to read an outstanding research paper, I highly recommend, Prospex, which was published at the IEEE Symposium on Security and Privacy in 2009.

This paper along with a lot of work that came before it and after it, have developed techniques to automatically recover network protocols from an executing binary program. There are also techniques to monitor serial and parallel port operations being done by running processes. Another tool that comes from the sysinternals suite called portmon, allows you to do this on Windows.

This used to be used for older malware when serial and parallel port communication was very popular. Then it sort of died out as no one used serially or serial or parallel ports anymore. But we've seen a resurgence in it recently, due to newer cyber physical malware that execute and infect much closer to the hardware.

Here's that example of WireShark that I was talking about. WireShark allows you to capture individual packet streams and pull those streams apart to inspect the source, destination, protocols that are being used, and even the payloads inside of packets. This can be extremely useful for getting a deep dive into how an executing program is communicating over the network.

Speaking of malware, there's some major advantages to using these dynamic analysis tools to monitor a malware's execution. First off, it allows for much easier analysis of encrypted, packed or obfuscated malware samples. When you are doing static analysis, reverse engineering any obfuscated or encrypted parts of a malware his body becomes very difficult, if not impossible.

This is by design, since it buys the malware author more time to keep the infection going and spread to more and more systems. But if you run that same malware sample using dynamic analysis or emulation techniques, where you can actually watch the execution run in real time, then you can step over or even avoid the encrypted parts, the packing and the obfuscation.

There are some techniques that allow you to do this such as running the malware and a debugger. Even IDA supported debuggers and Gidra as well has its own support for debuggers. And once the malware has decrypted or unpacked itself, you can even dump that unpacked or decrypted portion of the code back to an executable file on disk to be analyzed statically, like we've already learned.

## L3. Debuggers

>> Hello everyone and welcome back to Advance Topics in Malware Analysis, today we're gonna talk about using debuggers to analyze malware at runtime. That's right, debuggers, these are actually a fan favorite of reverse engineers, they are not just for debugging your buggy C code anymore. Reverse engineers use debuggers all the time to analyze malware samples at runtime.

Of course, a debugger can provide a disassembly, along the execution path that you're on. This is very similar to a static disassembly, except you're limited to only the code that's actually executing when you're looking at it in a debugger. It also allows reverse engineers to monitor and more importantly, modify the execution.

So when you're coming up on that call to the kill your machine function, you can replace this with a sequence of nops and not have to worry about getting into the dirty details. And continue to step forward in the execution of the malware. Debuggers also give you a nice readout of CPU registers, values in memory, flags, and the control flow.

You can trace accesses to specific memory regions and set breakpoints to avoid extremely fine-grained analysis. Just to get from one place to another in the malware's execution. There is a distinction here though, if you're dealing with kernel level malware, also known as rootkits, you're gonna have to use a kernel debugger.

If you're dealing with regular user-mode malware like we've mostly seen in this class, you can use any user-mode debugger to analyze the execution of that malware. For example, OllyDbg, all of the debuggers in IDA Pro and Ghidra, as well as GDB and WinDbg are all user-mode debuggers. These are often sufficient for any malware that doesn't specifically infect the operating system kernel.

If you are dealing with particularly nasty malware samples that have a rootkit component that infects the operating system itself, you'll need to use a kernel- mode debugger. Examples of these are the long gone softICE debugger, which was absolutely a favorite up until the end of its life, KD and NTKD, but nowadays, GDB and WinDbg actually are used most prominently for kernel-mode debugging.

That's right, they can do both. This allows you to analyze all of the execution, including all of the user level code, all the kernel code, and set low level breakpoints, that even the operating system kernel can't avoid. A little bit more on Kernel-Mode Debuggers. In the old days, you would have to connect the computer you're debugging to a different physical computer via a serial cable to inspect specific instructions and data with a kernel-mode debugger.

Thank goodness those days are gone. Nowadays, you can simply run a virtual machine on top of a host operating system. And then ask the virtual machine to expose a debugging interface and run a debugger such as GDB on the host and connect to the virtual machine. This allows you to inspect code running at any level of privilege in the virtual machine, kernel level or user level.

You can set low level breakpoints on specific kernel code that you want to break for any process that's running. An example of how I've used this reverse engineering malware was a specific malware I was looking into actually infected the way that windows were moving on the GUI of the Windows operating system.

I used a kernel level debugger to set a break point on the windows movement function. And then I could inspect the execution of the hijacked control flow whenever that breakpoint was tripped. This is what the process looks like in practice. You'll run an executable inside of a virtual machine, in this case, a Windows XP virtual machine.

And then ask the Virtual Machine Manager, in this case KMU, to expose a debug interface to the host. On the host, you can use a regular debugger like GDB, or WinDbg, or whatever your favorite debugger is to attach to that debugger interface. And you can now control everything that executes in the virtual machine as if you have your hands directly on the virtual machine's CPU.

I mentioned SoftICE and that's because, back in the day, SoftICE was always the debugger of choice. SoftICE was a kernel level debugger, that could actually run inside of the operating system that you were debugging. This was done by extremely intelligent low level tricks that faked the operating system into thinking SoftICE was actually running beneath it.

Unfortunately, SoftICE has been retired since this screenshot was taken. But this is an old case of reverse engineering a very old version of Skype using SoftICE. Interestingly, SoftICE was one of the only debuggers that could debug the Skype application. Because Skype contains many different anti-debugging tricks that are even impressive by malware analysis standards.

To get around these anti-debugger tricks, SoftICE was able to do it by pretending to be the operating system debugger underneath everything else. We'll talk more about how anti-debugger tricks work in the later slides. Then, of course, there's the Big Kahuna, GDB. GDB has been the default debugger on Linux systems for almost an eternity.

That's because GDB is just simply probably the most powerful debugger out there. What makes GDB so powerful is that it has so much functionality built in, but it's also infinitely extensible via plugins. You can get disassemblies, set breakpoints, even trace backwards on the stack to see what functions were called previously.

All of these operations require GDB to do some very heavyweight bookkeeping in the background of the process that's executing, and this is one of the main powers of GDB. Just how much data it can understand about the process that you're inspecting. Another fan favorite is OllyDbg. OllyDbg, provides almost all of the features of GDB, including a few extras that GDB doesn't have all wrapped up in a nice GUI.

It also runs on Windows by default. Unfortunately, support for OllyDbg has gone in and out, because the source code is often not being maintained. The old versions still run great and I still use them when I reverse engineer. Just a few basics of the power of OllyDbg, it's a user-mode debugger, and provides all the typical functionality you would expect from a debugger.

In addition, it supports several different kinds of breakpoints that are very difficult for malware to trick. Software breakpoints, such as writing int 3, or the CC opcode can be used an unlimited number of times. You can use four hardware breakpoints that actually use CPU registers to watch addresses to the executed and then break.

These can be much harder for malware to defeat. Olly also provides a memory breakpoint function where Olly will actually single step the program in the background. And watch for any accesses to that memory location. And when a memory access happens on that location only then will it break and report to you that that memory has been touched.

Very difficult for malware to defeat. Some more benefits of OllyDbg, for packed, or encrypted executables, you can isolate the decryptor, execute it, and then OllyDbg can actually dump the unpacked executable onto a file to be statically analyzed. Be careful though. If you're running a malware in a debugger like OllyDbg, it's very easy to get carried away and just hit run and hope your break points get hit.

You will very likely execute dangerous virus payloads. So you've got to be watching out for what's gonna execute next. Is debugging malware easy? No, of course not. Like any kind of modern malware reverse engineering, it's never gonna be easy. There are many layers of code obscuration, anti debugging and anti emulation techniques.

And even different techniques that malware can use to detect virtual machines, breakpoints, or if they're just running in an instrumented system. We're going to cover a number of these in the slides coming up. One thing I highly recommend is that you read chapter 17 of volume 3B in the Intel manuals.

This is going to give detailed information of how a CPU facilitates and has extensions for debugging. It'll give you a much better understanding of how debugging works at the lowest level, and the different tricks that malware can look for in order to detect or even subvert debuggers. In general, reverse engineering malware dynamically, especially when you're using debuggers and other high level tools is gonna require stepping back and forth between different tools to get a high level idea of what's going on.

Before you dive in and do fine grained analysis of the malware sample itself. I'm gonna give a concrete example of doing that exact process for a malware that I reverse engineered recently in the next lesson.

## L4. Real World Malware Investigation

>> Hello, everyone and welcome back to Advanced Topics in Malware Analysis. In this lesson, I'm gonna tell you about a real world malware investigation that applied some of those dynamic analysis techniques we've been talking about. In this investigation, it's some reverse engineering that my team did to unlock a ransomware sample without having to pay the Russians that appear to be behind the attack.

The case study here is actually one that began by watching the malware execute. You may recognize this screenshot as Procmon, the tool that we talked about before. What we did in this investigation was to fire up a virtual machine, fill it with what appeared to be realistic user files.

And then ran the sample that turned out to be RansomLock.AK inside that virtual machine. We watched what unfolded inside of Procmon and identified the victim files were being encrypted and left with a .crypted extension. The file extensions for the target files would be changed among different families. So this alone was not a good detector of the ransom lock family.

We did find, for example, that this family encrypted source code files like C, CPP, and header files, but other versions did not. We always hypothesized that this was because source code could be backed up in a repository, and so it wasn't a juicy target to encrypt by ransomware.

No families of this ransomware encrypted executables and this is because they want the system to keep running. As we'll see later, they're gonna pop up a screen asking the user to pay them in order to unlock their files. That's what gave us a clue how to unlock this ransomware ourselves without having to pay to get our files back.

To get a better idea of how Cryptolocker was encrypting these victim files, we opened up Cryptolocker in OllyDbg, the debugger that we talked about before, and executed it while also logging any library calls and returns using l trace in the terminal next to it. By watching these two, we could go back and forth between a high level dynamic analysis by watching the library calls and a very in depth dynamic analysis inside the debugger.

We could single step through the binary when we needed to, and eventually revealed that the malware had a baked-in statically compiled RSA encryption and decryption loop. This process was not fun in fact, it was quite tedious as malware analysis can be at times. But by switching between the high level analysis and the in depth analysis, we were able to identify that encryption loop relatively easily.

Finally, once we have figured out the encryption routine being used by the malware, and the key which was conveniently stored in the malware as body, we started to investigate the side effects of this malware that users could expect on their system. We found that a readme file was actually dropped on the victim's computer and automatically opened as soon as all their files were encrypted.

At the top of this file seems to be a Russian translation of the English text that's below. It asks you to email vladimirscherbinin1991@gmail.com to get a price to decrypt all of your files. This led us to believe that Vlad perhaps was 25 years old at the time that this malware was out.

Even better, the note said if you haven't heard back from the email address in 48 hours, please follow that link to fill out the feedback form on this malware. Of course, we had to follow the link, which resulted in this feedback form being popped up, where you can enter your email address and asked to have your files decrypted for a price.

You could even leave a personal note for Vlad, in case you had anything you wanted to convey to him. And even better, there was a CAPTCHA on this feedback form so that you couldn't spam the malware author. Sure enough, our reverse engineering concluded by hearing from Vlad who asked for $500 to get our files back.

Little did he know he was dealing with someone who is all ready a malware analyst and we had all ready figured out how to decrypt these files. Even more interesting, Vlad himself seem to have a Google+ page, rest in peace Google+.

## L5. Software/Hardware Breakpoints

>> Hello everyone and welcome back to advanced topics in malware analysis. This lesson is going to talk all about software and hardware breakpoints. So we've heard a lot about using debuggers in order to analyze malware. The key enabling feature for debuggers is the breakpoint. Software breakpoints actually use the int 3 instruction.

The int 3 opcode is just a one byte instruction that's specifically designed to support easy replacement of other instructions so that a debugger can break on any byte in a program that it wants. Debuggers replace the target instruction with an int 3 opcode, that is CC in hex.

Executing int 3 on the processor directly calls a system call, which looks up what process is being debugged and who the debugger is. And then the debugger regains control through the interrupt 3 handler. Detecting breakpoints is a common malware technique in order to stop you from using debuggers in order to analyze malware.

One of the best ways to do this is malware that simply search their own code section for the CC opcode. Malware can also keep a checksum or a hash of its own code to make sure that you've not inserted an int 3 instruction anywhere in the body of the virus.

If malware detect the int 3 opcode it can do all kinds of malicious things like remove the breakpoint and exit or even specifically punish the reverse engineer like destroying your hard drive. This is usually some obvious signs though for careful reverse engineers. You just watch for the malware to start behaving in ways that you wouldn't expect, like a debugger that just stops working or the malware just immediately exits or crashes.

Hardware breakpoints use special debug registers that are baked in to the processor itself. These debug registers are DR0 through DR3, and they contain the linear address of instructions where you want a breakpoint to be fired. A debug register DR7 is a status register that controls which of the breakpoint registers are set and some other status flags for debugging.

When the CPU detects that the instruction pointer now equals one of the addresses that's in the debug registers, it's going to raise that same interrupt. The same code that handles interrupt 3 will be executed, and that's going to transition execution back to the debugger. If you're interested in learning more about this, and a lot more fascinating features of the processor and how it handles debugging, you should definitely check out chapter 17 of volume 3B in the Intel manuals.

It'll almost blow your mind how complex modern processors are, and all the features they have to support debugging within the processor. However, malware of course have tried to figure out ways to get around this. Tricky malware can figure out ways to scan the debug registers and determine if the breakpoints are set on them.

And again, if they find that you're trying to debug them, malware can exit, perform some alternate behavior, or pretty much do anything they'd like. There are other more subtle ways of detecting that you're being debugged and malware have all employed these to get around being debugged. Using the read time stamp counter instruction is a common way that's almost impossible for a debugger to detect.

The rdtsc instruction reads the time stamp from the time stamp counter register. This is simply the number of ticks on the CPU since the last time the CPU was reset. This can reveal that you're performing dynamic analysis on a malware. The idea here is debugged or monitored code is always going to run slightly slower than nondebugged code.

So a malware can actually measure the passage of time using the rdtsc instruction. And then if it feels like you're taking too long to execute, the malware can then perform some evasive action. You can actually test this yourself by writing a little program that does nothing except for NOP instructions and then reads the timestamp counter.

Execute that program with and without breakpoints set on the NOP instructions, and you'll see huge time differences where the debugger itself started running. This is exactly how malware can use the read time stamp counter instruction to detect if you're doing any dynamic analysis. Another way to detect that a malware is being debugged is to detect the side effects of the debugger.

So for example, remember that great SoftICE debugger that we talked about earlier. Malware can actually detect that you're using SoftICE, because SoftICE has to replace a few interrupt handlers in the operating system kernel in order to handle interrupts properly. This is what made SoftICE so powerful. But malware can use this little snippet of code on the screen to actually check that SoftICE exists simply by checking the values in the interrupt descriptor table vectors that SoftICE uses in order to do debugging.

In these instructions, if you look carefully, the malware is grabbing the interrupt descriptor table, checking the interrupt 1 and interrupt 3 vectors, and seeing if their values make sense. If the values do not look like an untampered operating system kernel, then the malware can simply figure out that you're using a debugger and perform some evasive action.

Unfortunately for reverse engineers, Windows Operating System actually makes it even easier to detect that you're being debugged. Windows provides a number of different ways for a process to just detect if it's running in a debugger. The simplest way is to look at the PDB or the process environment block data structure that's in every process's memory space.

There's literally a flag there that detects if a debugger is present, and the operating system will always update that flag. Windows also provides an API, the isDebuggerPresent() function, that actually checks that flag for you. There are also other ways that the heap management is performed that get changed based on if a process is being debugged.

There are many different approaches on Windows that malware have figured out to discover if they're running in a debugger. And some of these are extremely obscure and quite interesting. You can read this article written by Symantec, the company that makes Norton Antivirus, about a whole list of anti-debugger tricks employed by malware.

## L6. Case Study Armadillo

>> Hello everyone, and welcome back. In this lesson, we're gonna be talking about some more malware that detect that you're using a debugger, and we're gonna wrap up the dynamic analysis lesson. We're gonna start this lesson with a brief case study of the Armadillo packer. We've talked a little bit about packers before.

Malware, or their packers, can include some very complex custom techniques to thwart debuggers. Armadillo, in particular, is a commercial packer made by a company called Silicon Realms. It's actually marketed as an executable compressor. It's widely used to protect expensive commercial software from being reverse engineered. However, its complexity has made it a favorite among malware in order to prevent reverse engineers from understanding how the malware works.

Some of the most amazing features that come prepackaged with the Armadillo Packer are double process debugging tricks, magic jumps, and full body encryption. We're gonna talk about these techniques right now. The double process debugging trick is a very slick way of preventing reverse engineers from attaching a debugger to the malware process.

The idea here is that most operating systems only allow for a single debugger to debug a single debugee. There's no reason to have multiple debuggers on a single process or multiple debugees for a single debugger. So the solution that Armadillo packer has come up with is to use two processes to debug each other at runtime.

The way this works is the first process as soon as it starts will spawn a child process. The child process will then attach to the parent process as a debugger. And the parent process will attach to the child process as a debugger. In this case, both processes are debuggers and they each have their respective debugging.

This prevents a reverse engineer from ever connecting a debugger to either process, because the operating system simply says operation not permitted because you can only have a single debugger debugging a single the debuggee. Armadillo goes a step further to implement magic jumps using this debugger-debugee relationship. These magic jumps are actually called nanomites.

And they're a brilliant twist on the double process debugging trick. Every jump instruction in the original binary is replaced with an in three software breakpoint opcode. The parent handles software breakpoints that occur in the child process and then looks up at runtime, the actual jump target in an encrypted jump table.

Once the jump target is identified, that section of code is actually decrypted live at runtime. And then the parent patches the child codes execution and resumes the child's execution. This way, when you open an Armadillo packed binary in IDA Pro, you actually don't see any control flow transfers at all.

You just see a huge block of straight line code with int 3 instructions everywhere that you would expect there to be a jump. This makes it extremely difficult to ever understand the actual control flow of an Armadillo packed binary. Defeating nanomites is extremely tricky when you have to unpack one of these samples yourself.

When your reverse engineering samples like this, you need to repeatedly and carefully terminate either the child or the parent in the debug relationship. Let it execute a little while and then dump the memory. Once you've dumped the memory, you can see what portions of the execution have already been decrypted.

And you can keep the real jump targets noted down to patch in the binary later on. If you do this enough, even though it's very tedious, you'll get a clean disassembly of the binary that you can reverse engineer. If you wanna read more, this is a great white paper written by a reverse engineer who actually defeated the nanometer technology.

Great packers like Armadillo are always in a constant war between reverse engineers trying to crack the packer and packers getting better and better. At present, evidence points to all versions of Armadillo being cracked even though there are very tedious to do and practice. An excellent read is this virus bulletin article, which contains an enumeration of many different anti unpacker tricks, being used by malware in the wild.

To combat anti-debugger stuff, reverse engineers are gonna have to patch the malware binary. And very slowly look out for different techniques that are gonna trip up your reverse engineering. Beyond the look out for double process debugging like we've talked about, the debugger detection tricks, breakpoint tempering, or even timing APIs, or the reed time stamp counter.

Malware will sometimes even insert illegal instructions into the pipeline, which will simply crash a debugger trying to follow along with the execution. You also wanna be careful of malware that are using interpreters. These are what's called VM packed or interpreter pact malware. Basically the malware body itself is just by code that's being interpreted by something like Python or a Java Virtual Machine.

You probably wanna keep the interpreter intact so that you don't have to reverse engineer the entire interpreter as well. And that's just a few of the anti-debugger tricks that you may run into when reverse engineering malware. The difficulty is gonna vary depending on the malware sample or the packer that you're trying to reverse engineer.

If you're interested, I highly recommend reading this research paper, published at X sack a few years ago, it discusses a completely invisible process for getting malware to unpack itself. IDA Pro, or Ghidra, both integrate debuggers natively, and this is a very powerful trick you can use when reverse engineering malware.

IDA knows that you can perform static analysis in IDA natively and dynamic analysis in a debugger. So you can combine the power of IDA and an integrated debugger to actually collect new knowledge via dynamic analysis. And it will automatically update all of the static analysis power that IDA or Ghidra have built in.

IDA and Ghidra both integrate many different debuggers, and plugins can add support for even more. The most powerful IDA Pro plugins you'll find actually manipulate both the static analysis and dynamic analysis automatically. An example of an extremely powerful IDA Pro plugin that combines static analysis and dynamic analysis with a debugger is this one that we've used to actually reverse engineer the BIOS code of a Windows operating system kernel during boot.

IDA can connect to the debugger, and supply all of the debuggers functionality back to IDA. This is a remote debugger. Debugging the operating system kernel as a kernel mode debugger, like the ones we talked about before. IDA can watch the execution as it steps through the booting code in order to debug malware that boot even before the Windows operating system.

Another very powerful feature of modern IDA Pro Plus debugger, or Ghidra plus debugger plugins, is the ability to do trace and replay. Trace and replay actually stores the entire execution state with every instruction that gets executed. This allows you to replay the execution exactly as it unfolded a previous time.

This can be very helpful for repeating analysis of very tricky malware samples. For example, you can record a malware unpacking itself, and then avoid having to do that miserable unpacking a second time around. In research, we often use trace and replay for differential analysis, to understand what paths differ when you give different arguments to a malware sample.

We'll talk a lot more about tracing in the next slide set. And finally, instruction emulation plugins for IDA Pro and Ghidra, both allow you to emulate the execution of regular code for small portions of the binary. This can also be extremely helpful for reverse engineering packed samples. It allows you to safely step through the code and emulate the values in memory, the registers and on the stack without actually executing any of the APIs, so it limits the danger that a malware can actually infect on your system.

This allows you to automatically watch how a malware would execute, but with some dummy values in the registers. For example, let's say you've identified a malware whose body is encrypted. You have found the decryption loop, and now you want to run only those instructions to get the malware's encrypted payload to spill out.

An X86 emulator plug-in would be a great thing to use in this scenario. Because you can simply point the emulator to the first instruction in the decryption loop. And you can set a breakpoint at the next instruction that gets jumped to after the decryption loop finishes. You can then tell the decryption loop to run in the emulator until the breakpoint is reached.

Once the breakpoint gets hit, then you know that the body of the malware has been decrypted, and you can continue your static analysis from there. And in summary for this lesson, we've learned how to integrate different dynamic analysis techniques to reverse engineer malware samples. We've identified data being transmitted or received by an application specifically the malware itself.

We've talked about malware sandboxing techniques and how you can leverage them to actually run malware samples to get a deeper understanding of what they're doing. And we've talked about the upsides and the downsides to using debuggers to analyze malware at runtime.