The Deep Dive Malware Analysis Approach

agdcservices.com/blog/the-deep-dive-malware-analysis-approach/

By AGDC Services

Today, we will review the primary approaches to malware analysis. Each approach is discussed and compared to one another to try and understand when you should use each method and why. We will show why the deep dive analysis approach is generally the most optimal and spend most of our time discussing this methodology. You should come away with the knowledge of how to appropriately apply the deep dive analysis strategy to any file and tips on how to learn this skill even if you have limited training time.

Analysis Goals

Before deciding on an analysis approach, we must first understand the goals of malware analysis. There are numerous goals possible. On some projects you will want to accomplish all of the goals, and others you will only focus on only a subset. The general goals are as follows:

- Identify Indicators of Compromise (IOC)
- Develop advanced network (SNORT) signatures
- Understand the malware infection impact
- Develop Yara signatures
- Obtain complete and accurate analysis results

These goals can be accomplished in any order, but they are listed in the most efficient order. IOC's are necessary for almost every analysis project and typically are the highest priority. Developing high fidelity network signature comes next because it can help with protecting the networks from the malware even if the malware network infrastructure changes. It can also help in providing attribution of the malware if you find new network infrastructure being used which is already attributed. Next you often want to understand the malware infection impact. This isn't identifying an exhaustive understanding of all the malware capabilities, but more of an overview of what the malware could have done within the network. For example, does the malware contain spreading capabilities, can it exfil files from the system back to the Command and Control (C2) server, etc. The next goal of an analyst is often to develop Yara signatures. This comes after understanding the infection impact so that you can design the best signatures. Yara signatures should ideally be built from code blocks which are difficult for the actor to change. If you perform this step before reviewing the majority of the binary, you won't identify the most unique code blocks to use in your signature. By waiting until after you skimmed over the entire code base of the malware, you can choose the spots in code which will be the most painful for the actor to change, leading to the best Yara signatures. Finally, your last goal is to obtain complete and accurate analysis results. This goal is not performed all of the time. Very often, an 80% solution in half of the time is better than a 100% solution which takes twice as long. This goal involves filling in any gaps that you came across previously. When you were reviewing the infection impact, you might have made some educated guesses that you weren’t 100% confident about. This step involves reviewing those guesses to make an absolute determination. This goals is typically only performed by analysts who are writing in-depth analysis reports on the malware.

Alternative Approaches

There are three primary analysis approaches:

- Triage analysis
- Focused Debugging
- Deep Dive Analysis

Triage analysis generally doesn’t review any actual code, or if it does, it is done minimally. This analysis approach usually performs environmental monitoring. You will set up a known environment, run the malware, and observe any changes to the system. It is a combination of performing an environment baseline comparison and physically watching for indicators such as network call outs in real time.

The Focused Debugging approach is centered around debugging. You may perform some simple analysis on the malware characteristics to identify expected behaviors, e.g. view the the strings and imports, but then you will run the malware inside a debugger to discover the details. The method revolves around knowing which library functions, or API’s, are typically used for different purposes and putting breakpoints on those functions. When the breakpoints are hit, you can observe the arguments and memory dumps to capture the IOC’s. You generally are only doing minimal reading of the assembly code. You will read the code around the functions of interest, but usually not outside of that. The focus is on debugging, identifying where in the code you want the malware to get to, and how to make it get there. This approach is much more advanced than triage. You can work around most anti-debugging tricks and also identify additional indicators that may be in memory, but only used as a backup, etc. These additional indicators will often be missed by triage analysis.

Deep Dive Analysis is the last approach. This approach centers around static analysis, i.e. reading the assembly code. You also use debugging, but only in a targeted manner. Your goal is to treat the assembly as source code, no different than if you were reading C or Python code. Your general understanding of what the malware is attempting to do will come from static analysis. The actual details may come from targeted debugging. For instance, reading the assembly may indicate a function is decrypting a URL to communicate to. You may not spend the time to statically reverse engineer the decryption routine, but instead you could use the debugger to simply decrypt the string of interest to uncover the URL. What the malware was doing came from static analysis, the details came from debugging.

Typical Usages

Now that we understand the three main approaches, when is each approach typically used?

Triage analysis has two main use cases, for incident responders where an infection is ongoing and IOC's are needed quickly to stem the bleeding, and for newer analysts who are still learning and don't know enough yet to dive into a debugger. Triage analysis often will only get the surface indicators and may miss things like backup C2 domains, be defeated by anti-sandbox techniques, etc. This approach really isn't appropriate when higher confidence results are needed, but it does have its place.

Focused debugging is probably the approach used by the majority of experienced analysts. You are able to somewhat quickly find most of the major indicators and you have the opportunity to identify and circumvent anti-sandbox techniques. It takes a bit more practice and education to learn what API's are important, how to read some assembly code, and to understand common programming patterns malware uses so that you know where to focus your debugging on. If you were to compare this to typing, you can think of this as the "hunt and peck" type of typing where you use two fingers. With some practice, you can type at a decent speed. You will never approach the speed of someone who uses all ten fingers, but you can get up to a decent speed much quicker with minimal practice.

Deep Dive analysis is the last approach. This is used by a smaller number of experienced analysts, most often only those who need to perform exhaustive reviews of the malware to put out reports. You can identify all IOC's with this methodology and will easily identify and work around any anti-analysis techniques used. For our typing analogy, this approach is compared to a typist who uses all ten fingers and has put the time in to practice and reach the top speeds.

The Dilemma

Now we come to a dilemma. We noted that focused debugging is used by the highest number of experienced malware reverse engineers, but we also contend that the deep dive approach is the most efficient. The question is "does common usage translate to the most optimal choice"? Our answer is a resounding no. There are reasons focused debugging is common, which we will get into, but we will also demonstrate why deep dive analysis is a much more efficient approach to use.

Focused Debugging Advantages

Focused debugging is so prevalent for two main reasons.

The first reason is debugging is relatively easy. Your focus isn't on fully understanding what's going on, but only trying to find the target areas where you might find IOC's. Once you've learned the main, high-valued API's, focused debugging is almost like a find feature. You simply put breakpoints on all the high-value API's, run the debugger, and wait until a breakpoint it hit. Once you're there, you examine the surrounding code / memory and extract IOC's. You don't need a complex strategy or full understanding of what happened. You just need to know the main breakpoints to set. There are only so many API's that malware commonly use, so even if you are just guessing, you can often get to the answer.

The second reason is focused debugging gives you answers which are "good enough". They may not be 100% complete, but what you have probably isn't wrong. You can think of your answers with this strategy as the 80% answer. Your results are going to be useful and you won't be embarrassed with your output, so many analysts don't feel the need to go further than that. If you can block 80% of the malware with relatively minimal training, that becomes a pretty popular strategy.

Deep Dive Advantages

Now that we know why focused debugging is so common, what makes deep dive analysis so optimal? Why should you choose this method?

The deep dive approach is generally the most optimal methodology you can use to find all of the IOC's in the shortest time frame. Deep dive analysis relies heavily on reading the code through static analysis. Static analysis is going to be far quicker and get more accurate answers than just debugging. Additionally, you can read and understand functions statically that you either can't reach with a debugger, or are very difficult to get to. That's a good reason to choose the deep dive approach, but what's backing up that statement?

Ultimately, malware analysis is just source code review. Only instead of reading C code, you're reading assembly. So let's ask our original question in a slightly different context. If someone gave you source code to analyze and determine its functionality, would you use triage, focused debugging, or the deep dive approach? In this situation, the answer is obvious. You would read the code statically, and parts you didn't quite understand, you might run in a debugger to examine the memory contents. That is the deep dive approach. If you tried to debug the entire program to understand its functionality, it would take forever and you'd probably be out of a job.

Another reason that supports the deep dive methodology is really a point against focused debugging. Debugging by itself provides an answer without context. Without the context of where you are in a program and how you got there, it's very easy to misinterpret the results. You can also easily miss indicators. Because you are focused just around the high-value API's, you may not have realized that there was an alternative path just prior to that high-value API which led to a backup indicator. Or maybe you're in the backup back and completely missed the primary indicator.

The biggest reason deep dive analysis is so efficient is that you can use tools optimally. You're not limited to static analysis or to debugging, you have a mastery of each and use either of the tools where it makes the most sense. You will spend the majority of your time reviewing the code statically to understand what is going on. Understanding a capability is generally independent of the arguments. For example, it is very easy to statically review a function to see that it is searching a directory to delete a file. You may not know what file is being searched for, but that doesn't impact your ability to recognize a file deletion capability in any way. When you need the actual indicator of what file is being searched for, you can switch to the debugger to view the memory contents to see the file path being searched. That is the power of the deep dive approach. You are never blindly searching the debuggers memory. You are always using it in a targeted manner to identify an item of interest. The flip side of using the debugger in a targeted manner is there isn't a need to step through the entire program in a debugger. You can jump around to just where you want to debug without needing to have the debugger follow the full normal path to get you there. That will save an enormous amount of time.

The last benefit is that you don't need to worry about VM snapshots, save points, etc. Because most of your time is spent reviewing the code statically, you don't have to worry about the state of the memory. All you are doing is marking up the code with your comments to make it easier to understand. If you are stuck on one file, you can just open up another to review, taking a break from the first. There is no need to revert your VM's to another snapshot, worry about accidentally reverting a machine and losing your work, etc. It sounds like a small thing, but it's surprising how much the wasted time of reverting snapshots and trying to remember which snapshot was in what state adds up to. In addition to the wasted time, it also takes you out of the analysis mindset and you lose a little extra time trying to remember exactly what you were doing prior to reverting, etc.

To summarize, the deep dive approach is the really the approach you use to review source code. Our goal is to treat assembly as just another programming language, no different than C or Python. You will want to apply the same techniques you use to review source code to reviewing malware.

The Disconnect

If the deep dive approach is so much more efficient than focused debugging, why do the majority of experienced analysts use the focused debugging methodology? Where's the disconnect?

The answer is pretty simple. The deep dive analysis approach has a much steeper learning curve before it becomes more effective. Essentially, it's the static analysis learning curve. Reading assembly is no different than learning to read any other foreign language. You have to spend hours learning the alphabet, then learn to sound out words, and eventually you will be able to read. Once you can read, you still need to build your vocabulary. If you never put in the practice, static analysis will be error prone and inefficient. But if you put the time in to practice, you will learn to read the language without mistakes, and finally you will learn to skim. Once you can skim assembly code, static analysis becomes really effective. Conversely, identifying high-value API's for focused debugging can be done very quickly with minimal practice.

To further compound the problem, most jobs don't provide the time for you to spend practicing static analysis. Answers are demanded immediately and as long as you get an adequate answer, it doesn't matter if there was a better answer that you could have provided if you only had the training. Given the time pressures most jobs put on analysts, they are often forced into the focused debugging approach whether they want to use it or not.

The last disconnect is that many think static analysis is only needed if you are required to understand every aspect of the malware, e.g. provide a full in-depth report. That's an unfortunate misconception. It misses the fact that static analysis can be tailored. Just like you can skim a book to find key elements, you can skim assembly code to find the indicators, to prioritize what to review, etc. The misconception comes from the steep learning curve required. Analysts believe that because it is so slow in the beginning, it must be that slow all of the time and only appropriate when you have to really dig deep into the malware.

What Deep Dive Analysis Isn't

Let's dispel a few of the common myths about static analysis. Many think static analysis is digging down deep into the assembly to understand what every instruction is doing, understanding every possible path through the binary. Static analysis is not "viewing the matrix". You aren't concerned with reviewing every instruction. You aren't even concerned with reviewing every function. You don't need to understand every line of code to understand what's going on, you only need to understand the key elements.

What Deep Dive Analysis Is

Now we now what deep dive analysis isn't, what is it then? Ultimately, deep dive analysis is source code review. It's a mix of static and dynamic analysis, but generally more focused on static. Static provides the what, dynamic provides the details. Deep dive analysis is looking at the binary holistically to understand the file at a targeted level. The key here is targeted level. You can target a full in-depth level to understand every function, or you can target a triage level to just understand the basic program flow. You also want to always keep in mind that you should be viewing the file holistically so you have the full context. You should never carve out specific parts of the file to look at individually. You lose so much context by not understanding how the target component fits within the rest of the file. You're likely to either make mistakes, or spend much more time than is needed if you would have stepped back and looked at the target component as just one part of the whole file. Deep dive analysis should always be looking at the file holistically so that you have the largest amount of information possible to help with your analysis.

The Deep Dive Analysis Algorithm

Finally onto the deep dive algorithm itself. We've talked around it, discussing some benefits, misconceptions, what it is and isn't, but we haven't defined the actual steps yet. The approach can be visualized by the graphic below.

The first step is to identify the main capabilities function. This may not be the actual main function the actor wrote, but could be a sub function within the main. First, we have to remember that the address of entry point in a file isn't the main function the actor wrote. The compiler adds code prior to the actor main which allocates some memory, initializes global variables, parses the command line, calls the actor main, and finally exits. Sometimes your disassembler will recognize the actual actor main and bypass the compiler generated code, but sometimes it won't. So first, we have to parse the compiler generated startup code to find the actor main. Next, we evaluate the actor main to determine if it contains the main capabilities loop of the file. Often times it will, but other times the actor main will perform some trivial capability or call a sub function or thread, and that sub function will contain the bulk of the malicious code. We are interested in analyzing the malicious capability and aren't concerned with a trivial function that only spawns a thread, etc.

Now that we have identified the main capabilities function, we want to build an outline of the main program flow. Remember that we are trying to understand the main program flow at an outline level, not an in-depth view. At this stage, we simply want a map of how the program works so that we know what components make up the file. Is a configuration file read, persistence set, networking activity performed, C2 commands executed, etc. You want to spend a minimal amount of time in this step. Just long enough to understand enough about each primary step of the file so that you can decide what you want to dig into during the next step.

With an outline of our main program flow, you now have a map of the file. The next step is to decide if further reverse engineering is needed. If you're just doing triage, more reverse engineering may not be needed. An outline of the program gives you enough information to identify approximately where the IOC's will be performed so that you can view the code and extract the IOC's.

If we want a more in-depth understanding, we move onto the next step and choose an area from our outline to dig into. You can choose any area to RE further, but a few primary areas of interest follow:

- Determine persistence methodology
- Identify network protocol

- Skim C2 commands to get general idea of impact of infection
- Perform in-depth analysis of C2 commands
- Identify optimal areas to build Yara signatures

Also note that while any area can be reverse engineered in any order, different goals can make certain orders of analysis make the most sense. For example, if you have limited time but want higher fidelity IOC's, it would make the most sense to skim the C2 commands prior to trying to generate Yara signatures. Because of the limited time, you probably won't want to perform an in-depth review of the C2 commands, but if you don't spend a little time to review the C2 capabilities, your Yara rules will be much more limited. Yara signatures are the most effective when they are built off of code blocks which are difficult for the actor to change. By skimming the C2 commands, you can quickly look for unique code blocks in each command function that would lead to high fidelity Yara signatures and be very difficult for the actor to circumvent.

Once you've identified the area to reverse engineer further, the next step is obviously to reverse engineer the target area. Once you've finished the analysis, you go back to your choice of deciding what area you want to reverse engineer next.

After you've analyzed all the areas of interest, the next step is to actually collect the IOC's. In the previous steps, you've identified all of the locations where IOC's would be generated. You know where persistence is set, you've worked through the network protocol, identified optimal areas for Yara signatures, etc. Now you actually turn that information into IOC's. You convert the network protocol into SNORT signatures, build the actual Yara signatures, etc.

The last step in the deep dive methodology is to report your findings. If you analyze a file and don't report the findings, what's the purpose of the analysis? Like every other step of the deep dive process, the important part of this step is to tailor it. You don't always need to produce a full, publishable, report on your findings. If you are performing triage, combining the indicators in a readable format along with the associated file details may be enough. If you performed an in-depth review, you may need a multi-page report. The goal of this step is to combine whatever findings you have come up with into a releasable format that can prevent another analyst from needing to duplicate your work.

Warnings

Let's discuss two caveats that are key to effectively performing deep dive analysis.

The first is that your understanding of what is happening should come from static analysis. If you are using a debugger to understand the what, you aren't using the approach correctly. What the file is doing, whether it's searching a directory, performing network communication, setting persistence, etc. doesn't depend on the inputs. You can identify network communication without knowing what the final address is. You can identify a directory is being searched to delete a file without knowing which file or directory is being targeted. The debugger is orders of magnitude slower than static analysis. If you are spending time in the debugger trying to identify the what, you are wasting huge amounts of time for no reason.

In some ways, deep dive analysis is fairly structured. You need to take a step back and identify your goals instead of trying to figure out everything at once. Most often analysis is done iteratively. You will make a first pass statically to identify an outline of what's going on. Then you will make another pass on the areas of interest to dive in deeper. Now you will have a solid understanding of what's going on. Then you use the debugger, if needed, to view the memory and identify actual values. If you try to do all those steps at once, you wind up going too deep into the weeds and it makes understanding much more difficult, and as a result, much more time consuming.

The second caveat is that dynamic analysis should always be targeted and be used only for verification and to identify memory components. This doesn't mean you may not have to rely on the debugger in the beginning while you are learning. But your goal is to increase your skills so that the debugger is only used for its intended purpose; to step through a program viewing and manipulating memory. Debugging should be used in any situation where you want to know a specific value, but are having a hard time identifying statically. There may be an API call that is dynamically resolved, but the API name is encrypted. That's a perfect situation to rely on the debugger. You may need to see a list of URLs which are being randomly chosen. The debugger is a great tool to examine the memory and view the entire list without having to wait for the program to randomly select every possible URL.

Insights

Here are a number of insights to help you better perform deep dive analysis.

The first insight is perhaps the most important. You always want to strive to analyze the file just as the programmer wrote it without jumping around. Malware is a computer program, and computer programs have a logical flow. Malware is no different. By following the logical flow of the code, you have the most context to help you understand what's going on. If I were to give you the number 32 and ask what the next number in the sequence is, you couldn't answer. But if I gave you the sequence 1, 2, 4, 8, 16, 32 and asked you what's next, you'll likely pick 64. You want to use that same context when analyzing malware. If you jump around the code, you lose all of the context and it becomes harder to understand what's going on, harder to understand the importance of variables, etc. By reading the code in the logical order, you can make educated guesses about what should come next. It's always easier to confirm your suspicion rather than blindly determine what a random piece of code is doing.

Another insight is the goal is to understand a function's purpose, and that should always be kept in mind. If you're reading a book and you don't understand a specific word, can you still understand the sentence, the paragraph? Most often yes. And the same is true with understanding what a function is attempting to do. You don’t need to focus on every instruction within the function. Look for key junctions and control flow to understand what’s going on. You may see a loop that is comparing two string values with an eventual exit condition. Often, all you need to understand is that you’re attempting to find a target string. It won’t matter what the string is, or possibly even what the string represents. The important note is that you are searching for a target string. There is a high likelihood that some other part of the function will provide context as to what the string represents.

That brings us to our next insight. A functions purpose is primarily identified through library calls. In windows files, API calls perform all of the heavy lifting. Probably 95% of all functionality is performed through a call to a library function. By understanding what API calls are used, in what order, and with what control logic, you basically reverse engineered the pseudocode of the function. Most of the assembly instructions are used to support the API calls They move around the variables in memory, build appropriate strings, setup the API arguments, but they rarely do meaningful work. The primary use of the assembly instructions are to get the environment setup for the API calls.

There are some exceptions to this. Library calls can be statically compiled into the binary and be unrecognized as an API call. The API call can be inlined with the actual assembly code. And a select number of functions can be manually performed straight through assembly instructions. These are all the exceptions though. 90% of recognizing a functions purpose is by matching the order the API calls are performed to a known algorithm. If you aren’t able to recognize the 10% of the exceptions listed, it means you will be missing some of the library calls, which will make identifying the algorithm more difficult. Not impossible, but definitely harder.

An important insight is everyone starts off thinking malware code is very sneaky and will hide what it’s trying to do, send you down false paths, etc., but that’s not the case. Malware rarely tries to hide its purpose. If you see a function that looks like it’s performing a specific capability, it probably is. If you don’t quite understand how the program flow gets to that function, just know it probably does at some point. Malware authors don’t spend a lot of time trying to trick the reverse engineer. There are exceptions to this, namely packed code and obfuscated code, but that’s more of an exception. Packed code is more geared towards hiding from AV scanners. Once you unpack the code, the functions are straightforward like all other malware.

The one true exception is that some malware will actually be put through an obfuscation program. This could add a lot of garbage code that doesn’t do anything, add calls that aren’t germaine to the program’s flow, and even add whole functions that never get used. This is a solid exception to the rule, but luckily only a very small percentage of x86-64 files uses this technique. Managed code, like .NET files, where you can decompile the code directly back to the original source code will be put through obfuscation programs more as a rule than the exception. But x86-64 code where you are looking at assembly instructions rarely implement the obfuscation techniques discussed here.

A critical insight, somewhat obvious on the surface but which has a deeper meaning , is the more functions you recognize, the less guesswork is needed. Malware analysis is a mixture of art and science. You often won't recognize every line of code, every function. There will be gaps in your understanding, but you still need to make a determination of the intended purpose. A good analogy is solving a jigsaw puzzle. Science lets you turn over the puzzle pieces. The more pieces you have turned over, the easier it is to recognize the picture. What order you turn the pieces over, how you identify the puzzle when you have a pixelated picture with blank spots is the art. This insight is specifically geared towards learning to recognize common library functions which are optimized by the compiler.

There are three main ways a function can be included in a file, through an external API call, statically compiled, and inlined. An external API call is easy, you will have the API name and there is no guesswork. A statically compiled function will have the entire library function placed into your file. These will often be recognized by your disassembler tool through a signature database, though not always. The third way is to be inline optimized by the compiler. In this case, instead of having a separate function which is called, the compiler takes the bulk of the library code and places it directly inline with all of the other assembly instructions. That means instead of having a call to a function which might be recognized and labeled as a library call, you simply have the assembly instructions. On top of that, some library functions will be optimized and different assembly instructions from what's in the original API function will be used. The replacement instructions perform the same capability, but in a more optimized manner. In both of these cases, because there is no actual function called, there will be no signature database to identify the function. From the disassemblers perspective, there is no difference between an inline function and regular assembly instructions. That means it is up to you to recognize the assembly instructions as a library function as you are reviewing the code. If you don't recognize the inline function, you will not have the complete information about what is being done inside the function you are reviewing.

As we previously stated, the bulk of the work inside a program is performed through API calls. So if you miss API calls because they are inlined and not recognized, you will be missing some portion of the work being performed. For our puzzle analogy, this means you will have more blank spaces in your puzzle. You may still be able to recognize the picture, but it could be significantly more difficult depending on what the inline functions are.

Our next insight is that static comprehension is primarily pattern recognition. Luckily for us, pattern recognition doesn't depend on matching a pattern line for line. You only need to match the key points. The key points for programming algorithms are the library calls and the control flow, i.e. the conditional logic such as if statements, for and while loops. The library calls are the most important. Often times, you can identify a functions purpose simply by the order of API's called. If you see the API's, OpenProcess VirtualAllocEx, WriteProcessMemory, and CreateRemoteThread without knowing anything else, you can almost certainly say the function is performing DLL Injection. To be more confident, you would check the control flow to make sure the API's are called in the appropriate order and all are used in the same control path. If one or two of the API's are used only in error conditions, then it may change the functions purpose. Outside of a few niche categories like cryptography and unpacking, the assembly instructions are there mostly to set up the environment for the API calls and don't add significant information for determining a functions purpose. The end result is 80% of learning to statically analyze a function comes down to learning a large set of programming algorithms and then comparing the API's you see in a function against your known algorithms.

Our last insight is about your tools. Disassembler signatures of library functions will fail at some point. If you pay for premium disassemblers and always stay current with the latest release, your tool will recognize all of the statically compiled library functions that are possible. But even then, it will not recognize all of them. Disassemblers recognize statically compiled library functions based on signatures. Unfortunately, there are just so many ways library functions can be changed by the compiler based on optimization settings and updates to the library code themselves. The disassembler will never recognize them all. What happens very often is that the tool signature will be updated for a new version of a library function. But then that library function will be updated more frequently then your tool and it will cease to be recognized.

Fortunately there is a silver lining to this. While tool signatures have a hard time recognizing slightly modified library calls, people do not. The small changes that disrupt a disassembler will probably go unnoticed to you. That means you want to use your disassembler signatures to manually learn to recognize statically compiled library functions so that you can recognize them yourself when the tool fails you.

This is very simple and doesn't need to take a lot of time. All you need to do is to take a quick scan of statically compiled library functions when they are recognized and labeled by the disassembler. Look primarily at the graph view, and then look to see if there are any identifiable constants or strange instructions that you don't often see. These aspects rarely change much between versions. Don't spend a lot of time trying to memorize these functions, just take a quick look. The more times you see the library call when you know what it is, the better the chance is you will remember it when you see it and the tool doesn't recognize it.

In this case, volume is key. If you only see the statically compiled rand function once, you probably won't identify it on your own later on. But if you see it 20 times over the course of 6 months, there is a very good chance your brain will make that needed connection and on the 21st time, you will be able to recognize it with or without a label. As we've mentioned, identifying the API calls is key to understanding malware capability. A lot of time and money goes into building disassemblers. Use that to your advantage to learn from them so that you can identify all of the possible API calls to make your job as a reverse engineer much easier.

Tips For Learning With Limited Time

Up until this point, we've discussed the different analysis approaches, identified why deep dive analysis is most optimal but why it's not the most common, and explained the approach along with warning and insights to make the best use of it. While the approach may sound great on paper, you may still be in the position where your job just doesn't provide the time to build up the static analysis skills necessary to make deep dive analysis optimal. To that point, we will discuss a few methods to build the deep dive skills with limited time to practice.

Our first tip is to get your answers first the way you know how, then review statically. People learn in funny ways. In the beginning, code will most often be pretty confusing when you look at it statically without already knowing what it's doing. If you spend time using any method you know of to understand what's going on, and then look back at it statically, it's surprising how much clearer the assembly will look. It's basically the same as reviewing a solution guide at that point.

You won't learn it as quickly compared to if you spent the time to really struggle through statically, but you can do this in much smaller chunks of time. If you just try to read unknown code statically, it could take you hours to understand a target function. If you don't have the hours to spend to get to the answer, you never really know if you were reading it correctly and it's not conducive to learning. If you can spend the hours to get a definitive answer, your brain will make the strongest connections and you will learn it in the shortest amount of total time. But that's only if you have the hours to spend.

When you don't have that long, it's much better to understand the capability dynamically, then review statically. It will be much clearer then. You will have to do this a number of times before you will make a permanent connection in your brain so that you can understand the code purely by static analysis. But when your job demands quick answers, it's often more feasible to spend 10, thirty minute chunks of time over a longer period reviewing code where you know the answer rather than spending 3 hours straight. The 3 hours will be less total time, but the 30 minute chunks of time are much easier to fit into a busy schedule.

This tip is critical. You have minimal time to practice. You have a small break and are motivated to dive into a piece of malware to practice your skills, but all you have available are simplistic samples that won't challenge you. Now your small training window has passed and you didn't get a chance to hone your skills. That's why you should always keep a training index. Each file you work, you should try to learn something new, dig a little bit deeper than the last time. But there will be many, many files where you don't understand everything. The crypto is a little too advanced, the networking protocol is too complicated. All of those files should be recorded in your training index along with a note of what interesting aspect they contain. Now you have another 30 minute window where you are free and want to get better at reversing cryptography routines. You pull out your training index and search on crypto and find half a dozen files that have crypto routines in them that you weren't quite able to understand. You choose the first one and spend your time digging into the routine. Your 30 minutes weren't wasted this time and you improved your tradecraft in the precise area you were interested in.

If you don't keep a training index, you will be at the mercy of whatever you have available when you have the time. That's not a recipe for success. You should always keep notes of what you worked on with the interesting capabilities. In the beginning, they can be used to practice. Later, once your skills have progressed, they can be used as examples to help teach others the skills. There is no greater way to learn a skill in-depth than to teach it to someone else.

The next tip is that it's quicker to learn programming versus learning to reverse engineer code. This is somewhat counter-intuitive at first. It's common to think that you must practice reverse engineering assembly to become a better malware analyst, but that's not 100% accurate. One skill, and the easiest, is reading the assembly. Once you're comfortable reading and skimming assembly, the more important skill is recognizing what programming capability the assembly is attempting to implement. And that comes down to knowing how to program a wide range of capabilities.

What that means is if you have limited time to practice, you will build your static analysis skills by learning a multitude of ways to program common malware capabilities. This can be done in much shorter blocks of time because there are far more resources dedicated to programming, so you can easily find good source code to study. The larger your knowledge of programming capabilities is, the easier it will be to recognize what capability the assembly is attempting to perform in malicious files. Often times, the difference between an intermediate reverse engineer and an advanced analyst is simply the breadth of capabilities they are aware of and are able to recognize.

The last piece of advice is an age old concept that applies to learning any new skill. Simply practice every chance you get. Don't be concerned with how much time you have or if you will make a breakthrough. Learning reverse engineering, particularly learning to read assembly, is very much like learning to read a foreign language. You will have to sound out a word X number of times before your mind will make the connection and you begin to recognize it on sight. You will sound out the same word dozens of times, making slow progress and the same mistakes each time. Finally, it will just click and you will recognize it. This makes practice frustrating because you aren't making constant progress. Your progress comes in steps. You will feel like you aren't making any progress for a while, then you will make a big jump forward. Then no progress for a while, then a big jump forward. Those small windows where you can take 15 or 30 minutes to practice may not seem like you are making progress, but it is instrumental to your brain forming the new connections needed to make that big leap forward. The bottom line is it will take many dozens of hours to learn this skill. If you don't take the time to practice, you will be forced to rely heavily on debugging and guessing at what's going on instead of simply being able to know what is going on.

Summary

That brings us to the end of learning the deep dive malware analysis approach. Let's leave off with a summary of a few key points we went over.

The deep dive approach is a mix of static and dynamic analysis, but more focused on static. The what should come from static, the details can come from dynamic. It's important to use the right method in the right circumstance to get optimal results. Don't fall into the trap of trying to debug everything because it's easier to understand when you see real values and can watch the exact code progression. Take a step back and remember the details don't change what's going on and the "what" really is 90% of what you need to understand. Once you know how persistence is being set, how the network protocol is set up, determining the details is often trivial. When you don't know those whats, you're searching for a needle in a haystack. You can find it with enough time and effort, but it won't be efficient.

On the flip side, if you're trying to identify a specific value that's being built in memory, static analysis may not be the best approach. If the value is being decrypted, concatenated with other values, copied around into different variables, then debugging is the right approach. The key is targeted debugging. Don't step through the entire program just to find one value. Use your understanding of how the program is set up gained from static analysis and jump right up to the point you need to observe the targeted value. When debugging, you shouldn't be single stepping too much. You should either be changing the EIP to get right to the location you need, or setting key breakpoints at a target location because you know the binary will reach that point. Combining static and dynamic analysis is where you will get the most efficiency in your approach.

Static analysis will provide you more accurate answers far quicker than debugging, but only after you get good at it. There is a far steeper learning curve to static analysis before it will be more efficient than focused debugging. If you want to reach the expert levels, you really have to get good at static analysis. It's easy in the beginning to think debugging is superior. Static analysis seems so slow and error prone because you're not good at it yet. It's hard to imagine it would ever be best. Think back to grade school when you were learning algebra. It is so hard in the beginning when you just have symbols and no real numbers to look at. You often would have to put in fake numbers just to understand what was going on. But if you really took the time to work on it and get good at algebra, you eventually got to the level where the actual numbers just slowed you down. You could solve more problems quicker with algebra than if you had to always fill in the numbers. Algebra is like static analysis and using discrete numbers is like debugging.

Don't get discouraged, progress is not linear. Take the time to practice and you won't be disappointed. It really comes down to belief. If you know the end result is you will attain a mastery that's not possible using focused debugging, it's easier to stick with the practice in the beginning. It will be discouraging. You'll feel like you aren't making any progress, but I can assure you that you are. Tell yourself everyday that progress is not linear. If you are putting in the time, you will be getting better. You may just not realize it. Keep a history of files you worked on and look back at them after 6 months and you will see how far you have come. What was incredibly complex 6 months ago will be much more understandable today.

The last thought I want to leave you with is to come back to comparing this deep dive approach to learning to read a foreign language. Focused debugging is similar to buying a travel phrase book. You will be able to ask where the bathroom is and get around a city in a relatively short order. It can allow you to visit the main tourist sites on your own. Deep dive analysis is like becoming fluent in the language. Now you can visit the tourist locations, but you can also read the signs to get more background information. You can talk to the locals and get tips on hidden gems that aren't in the guide books. You can travel the country outside the main attractions to see everything and not be limited just to the tourist traps. All you have to do is spend time to master this approach and you can understand everything about a binary, or you can quickly skim a file to find the IOC's and move on. The choice will be yours, but you will gain the knowledge to actually have that choice.