

Discussion

The final chapter of this project is dedicated to exploring the work that has been done in relation to scalability and automation. If the techniques proposed in this project can be applied to a broader scope of vulnerabilities, we imagine the result can be used to build a fully automated vulnerability detection application.

To highlight the work needed to reach the final goal, we will first compare function hooking and Event Tracing for Windows (ETW), explore exactly what information can be extracted from a patch, and hereafter use this information to discuss how the work done to detect CVE-2021-24086 might be applied in an automated and scalable way to detect other vulnerabilities.

1.1 Comparison of function hooking and ETW

In the previous chapters we have explored the world of Windows telemetry, analyzed a recent and critical vulnerability, and hereafter combined all of the knowledge gained to detect the vulnerability using the explored telemetry sources. In this chapter we will be exploring and discussing how the techniques used to detect CVE-2021-24086 possibly could be used to detect other vulnerabilities. Furthermore, we will also explore if, and possible how, we can automate the whole process to automatically detect exploitation of a large subset of all recent vulnerabilities given a patch.

To detect CVE-2021-24086 we explored both ETW and function hooking to gather relevant telemetry to discover a vulnerability in Windows. Unfortunately, we discovered that the detection opportunities, at least for CVE-2021-24086 and other vulnerabilities present in `tcip.sys`, are not great with ETW as most telemetry herein is based on entering *known* unintended states. With function hooking however, we get to define what is an unintended state and can therefore tailor our telemetry to our specific detection needs.

If we disregard the fact that ETW telemetry was not sufficient to detect CVE-2021-24086, we still want to compare the two methods in order to give the reader an overview of weaknesses and strengths for each method. The results of this analysis can be seen in table 1.1

	Function hooking	ETW
Cons	Complex	

1.2. PATCH INFORMATION

	No tolerance for error Might slow down the computer Kernel mode Long development time	Unable to customize events Lack of events Not able to create custom events Unable to detect CVE-2021-24086
Pros	Custom event filtering	
	Customize detection to specific vulnerabilities High level of control Able to detect CVE-2021-24086	Built-in to Windows Robust Fast User mode

Table 1.1: Comparison of function hooking and ETW

As function hooking and ETW is so fundamentally different, it is hard to compare them in a proper way. Function hooking has a high level of customization which also brings a high level of complexity. ETW, while still being somewhat complex, works out of the box, has a high level of robustness, but gives very little in terms of customization. Furthermore, ETW could only be used to detect CVE-2021-24086 after the patch was applied, which is not that valuable in regards to the scope of this project. Given more vulnerabilities, we do believe that ETW could show its worth, but it is impossible to say without further work analyzing more vulnerabilities. Perhaps a combination of function hooking and ETW is optimal, as both can be run at the same time without large penalties, both in regards to performance, security and scalability.

1.2 Patch information

In ?? (??) we analyzed and reproduced a vulnerability using only publicly available information and information gained from the patch. While publicly available information was very scarce, extracting information from the patch gave us very valuable and usable information quickly revealing the root-cause of the vulnerability. While developing a full fledged Proof of Concept (PoC) took a lot of time, revealing the root-cause of the vulnerability was very quick.

The goal of this project is mainly to discover the information given by a patch, and determine if this information can be used to detect the vulnerability in an unpatched system. Therefore, we must detail exactly what information can be extracted from a patch given the right tools such as BinDiff[5] and IDA Pro[2]:

1. Very precise byte by byte changes of the binary
2. Which functions were modified, including confidence and similarity analysis
3. Specific addresses where code was modified, added or removed

1.3. SCALABILITY AND AUTOMATION

4. Changes on the assembly level
5. Additions to the binary, such as new functions

Luckily for us, all security patches are distributed separately from feature upgrades, allowing us to only look at changes we know are related to security.

If we were to relate patch information to detection, it is important to highlight just how much information was gathered from the patch. First of all, using the list of modified functions, we were quickly able to locate the vulnerable function (`Ipv6pReassembleDatagram`). To detect CVE-2021-24086 without a patch or a reliable PoC, would ultimately require us to rediscover the vulnerability based purely on publicly available information. We deem this infeasible for us, and most likely many other security professionals, and certainly deem it infeasible to do on a larger scale.

When patch information is analyzed by someone with the right skills within vulnerability research, it gives tremendous value both in detection purposes, but definitely also within exploitation purposes. The question remains however, whether the right information can be gathered programmatically and in a way that is easy to scale and automate. The next sections will explore this further.

1.3 Scalability and automation

The most important part of scaling and automating the detection of vulnerabilities given a patch is to extract the patch information and interpret it correctly. The following sections will first explore how information can be extracted without human interaction and hereafter how to detect the root-cause of a vulnerability and use that information to create detection logic to detect it.

1.3.1 Information gathering

Automating information gathering is a somewhat simple task. The binary comparison can be fully handled by a tool such as BinDiff[5], whereafter the information can be extracted. BinDiff uses a undocumented binary format with the extension `.BinExport`, making the extraction of the information somewhat cumbersome. With that said however, the Ghidra[4] extension BinDiffHelper[1] has code to extract the necessary information from the binary format which we can most likely reuse without much effort.

1.3.2 Root-cause analysis

To be able to detect a vulnerability, knowing what caused it is a very important step. If we consider CVE-2021-24086, the root-cause was a null pointer dereference due to a length not being checked correctly. So while the root-cause is a null pointer dereference, the detection does not directly detect the null pointer

1.3. SCALABILITY AND AUTOMATION

dereference. The detection actually worked by looking for the symptom leading to a null pointer dereference. However, as the Windows kernel and privileged applications are mainly written in C and C++, many other vulnerability types are prevalent. An unexhaustive list of the most common vulnerability types can be seen here:

1. Null pointer dereference
2. Stack overflow
3. Heap overflow
4. Use-after-free
5. Type confusion
6. Arbitrary memory overwrite
7. Integer overflow
8. Logic bugs

As one can probably imagine, most of these vulnerability types are not fixed in the same way, and in some instances the same vulnerability type might be fixed in different ways as well.

If we examine one vulnerability type, such as integer overflows, we can see the complexity of analyzing the root-cause. Imagine a scenario where a buffer is allocated with the size calculated as the sum two 16 bit unsigned integers. The length of this buffer is stored in a 16 bit register as an unsigned integer. In cases like these the integer can overflow such that the buffer size will be calculated as such:

$$\begin{aligned} \text{bufferSize} &= \text{uint16}(u_1 + u_2) = (u_1 + u_2) - 65535 \\ &\text{When } u_1 + u_2 > 65535 \end{aligned} \tag{1.1}$$

In instances like these, the flaw might be fixed in two different ways depending on what part of the logic is flawed. In cases where the buffer is never supposed to be bigger than 16 bits a bounds check must be added. This is usually reflected as a `CMP` instruction in the assembly checking if the sum of the two 16 bit unsigned registers is larger than `0xFFFF`. If the issue lies within the fact that the buffer has the wrong maximum size, the 16 bit register could simply be swapped with a 32 bit register.

As the example highlights, finding the actual root-cause of a vulnerability is not easily achieved. For just one vulnerability type, at least two obvious root-causes with fixes can be found. Another issue is also the fact that at the time of root-cause analysis, we do not know the vulnerability type. This makes it incrementally more difficult to define the root-cause, as you are restricted to

1.4. FUTURE WORK

only basing your analysis on the changes in the binary. Additionally, the same binary often contains more than one vulnerability, as seen in the patch tuesday of July 2021[3] which patched nine vulnerabilities in Windows DNS Server.

1.3.3 Detection logic

The next step in detecting exploitation of vulnerabilities at scale is to create detection logic that will trigger on exploitation attempts as we did with CVE-2021-24086. If we assume that we are able to automate the process of obtaining patch information and analyzing it to correctly determine the root-cause, there is still a lot of unanswered questions in regard to creating the detection itself. Again, depending on the type of vulnerability, the detection needs to be created differently.

If we base this analysis on the integer overflow example as explained above, the detection can be somewhat automated. If we take the example where the patch contains a `CMP` instruction to ensure the size is no larger than a specific value (such as 16 bits, $2^{16} - 1$), we can simply have the same logic in our detection. In the other example with integer overflow, where the fix was to switch the register, a similar detection logic could be employed to detect when the overflow happens.

One unanswered question however, is how do we ensure that we check on the right value. As we have explored earlier, function hooking redirects the execution at the beginning of the function. However, some cases exists where the register values we base our detection on is not present in the beginning of the function. In these cases we have two options:

1. Trace the usage and calculation of relevant registers
2. Hook the function at the fix location

The first option where we track the usage of specific registers we know are involved in the calculation of the value we detect on can be a very cumbersome action. In CVE-2021-24086 we were able to calculate the value directly from memory, but in some cases we must also execute other functions and do advanced calculations, that might ruin the source function's execution flow. With the constraints explained here, it will only work for a subset of cases, such as it did with CVE-2021-24086.

The second options where we hook the function at the specific location also has certain constraints and issues. First of all, when hooking a function at the beginning, we only have to save the registers and stack involved in the calling convention as explained in ?? (??)

1.4 Future work