An Analysis on Windows 11 Defender with Novel Undetected Malware Architecture

Noah Ostle
School of Computer Science
University of Technology Sydney
noah.j.ostle@student.uts.edu.au

Samuel Smith
School of Computer Science
University of Technology Sydney
samuel.g.smith@student.uts.edu.au

***Abstract*—This paper [2] investigates the efficacy of Windows Defender in the context of a user who makes a fresh install of Windows 11, and leaves Windows' security features on their default settings (as is exceedingly common for personal computers).**

We evaluated Microsoft's default level of protection against the average user by first collecting previously used malware techniques, implementing and testing them, and analysing both if and why each technique is detected by Windows Defenders' most current version.

Then, we design our own payload deployment architecture with the goal of bypassing Windows Defender to execute malicious actions on a victim PC. From the effectiveness of our method, we may draw conclusions about the security of Windows Defender, and the implications this has on the average user traversing the current threat landscape.

I. Introduction

Malware is an issue that is currently plaguing both users and organisations alike, with the former generally having much less consideration for the hardening of their Operating Systems.

Thus, in this paper, we set out to explore just how well a typical personal installation of Windows 11 Home's latest version - 21h2 - fairs against modern malware techniques. Strictly speaking, we aim to see if it is possible to execute arbitrary malicious payloads against up-to-date Windows 11, without security alerts or actions, with Windows Defender and all other default Microsoft security features completely enabled.
As our hypothesis; *'Windows Defender's focus on signature-based analysis leaves it vulnerable to bypass tactics, allowing obfuscated malware to execute without triggering alerts.'* suggests, the objective of our research is to discover malware techniques that evade Windows Defender Antivirus, and evaluate what the success of these techniques implies about the security of Windows Defender and Windows 11 at large.

Our method of testing this hypothesis includes performing a literature review, including keyword searches and comprehensive topic analyses in order to collate past techniques that have been proven successful in the literature (albeit on past versions of Defender), reimplementing these techniques in C, evaluating their effectiveness against current Windows Defender versions, and combining, modifying, or extending them in order to create a most effective novel technique.

As such, the scope of our research includes an evaluation of previous techniques against Windows Defenders most recent version, miniature ablative studies on the reasoning for their success / failure to bypass Defender, the design and formulation of a novel technique which aims to exploit weaknesses of Defender found by our analysis, the testing of our technique, and finally an analysis on the implications that our techniques' success (or lack thereof) has on the security of Windows 11 devices.

We believe that our research is highly relevant, as if we succeed in creating a technique which performs well against our metric, we have essentially proved that current versions of Windows Defender, and by extension Windows 11 should not be considered secure against a motivated and skilled attacker. In addition, if we fail to find a successful technique, we have added confidence that current versions of Windows Defender do indeed provide adequate security measures to prevent skilled attackers from infecting victims with malware.

II. Evaluation Methods from the Literature

Briefly before we start on our Evaluation Framework, we find it important to note that due to several factors we are not able to use a 'field standard' framework or criteria to evaluate or compare out results with that of past papers.

Firstly, we find that based on both the relatively small amount of peer reviewed research into Windows Defender, as well as the lower than desired level of academic rigor associated with papers within the field of malware at large, that there is no stand-out, commonly used scientific framework for which we could base our analysis. Many of the papers we have used in our research perform a more *qualitative* analysis on the effectiveness of certain techniques, not to mention the fact that the evaluation criteria usually consists more or less of two scores; either 'Detected' referring to a bad score, or 'Undetected' indicating a good score.

Strictly speaking, due to the design of our experiment -

that we take the results from several papers, and retest them ourselves to determine which show promise against Defenders' new versions - that there is no single evaluation framework design that is common to all of the papers we would wish to compare our results to.

In addition to this lack of a rigorous scientific standard, it is also important to note the constantly evolving and changing world of security. It is true that the field of cybersecurity changes faster than most, with techniques or ways of thinking from less than 5 years ago considered out of date, making it sometimes counterproductive to try to follow an exact method from a past paper.

Due to this, we have decided that we will design our own new evaluation metric in an effort to construct a fair, modern, and logical framework by which our technique can be compared with the techniques from past papers under newer versions of Microsoft Defender. This framework will aim to fairly score each techniques effectiveness against Defender, in combination with several other factors we deem important for achieving stealth. We aim to do this in a way that fairly measures efficacy in real-world context, and allows us to examine the reasons why a given technique did or didn't work.

III. Framework for Evaluating Techniques

Now that we have identified several promising techniques from the literature, we may devise a framework through which we can establish the characteristics, and effectiveness of each technique.

Since we are strictly testing on the most modern version of Windows 11 - 21h2, simply extracting results from the literature is not enough. Therefore, our first step is to take each technique, implement it in C, and re-evaluate it against our chosen version. We may take the following considerations into account when designing a metric by which to evaluate each technique:

Primarily, our metric must be concerned with whether a given technique *is* or *is not* detected by Windows Defender. In our experimentation, we determined that a detection can occur in two stages;

Static Analysis, which is when a malicious file is saved onto the Disk of a computer, where Defender can perform file scans and signaturing in order to attempt to flag malicious activity.
and
Dynamic Analysis, where Defender attempts to scan the behavior of a file as it is running in order to try and detect malicious patterns - this is commonly done by pattern recognition of assembly code, as well as hooking and monitoring various functions and Win32API calls which are commonly used for malicious purposes.

To gain additional data on how well a given payload performs, we may also consider the case where a particularly security conscious user performs a *VirusTotal* scan before running an application.
VirusTotal is an online tool which allows us to upload a file, and have it statically scanned by 72 different Anti Virus programs. This will give us a good overview of how well our sample performs against a variety of different Anti Virus programs in addition to Windows Defender, and can help us get a more fine grain perspective on The effectiveness of different techniques.

We can use these three measurable outcomes to compose our metric, giving each a weight over the total score as follows;

Component	***Rating (r)***	***Weight (w)***
Defender Static Analysis	$\begin{cases} 0 & \text{if detected} \\ 1 & \text{if undetected} \end{cases}$	$\frac{2}{5}$
Defender Dynamic Analysis	$\begin{cases} 0 & \text{if detected} \\ 1 & \text{if undetected} \end{cases}$	$\frac{2}{5}$
VirusTotal Scan	$1 - \frac{\text{score}}{72}$	$\frac{1}{5}$
Total		$\sum r \cdot w$

TABLE I
Evaluation Metric

We see that under this metric, detection by Defender has the primary effect on a techniques score, but we have also included VirusTotal score to provide differentiation between techniques that rank the same against Defender.
This is a good starting point for comparing our techniques, but as we will see later, these three factors do not encompass all the behaviour we might want to measure. Thus, it is important to keep in mind that there will be additional factors we may take into account when evaluating the success of a technique - henceforth referred to as our 'heuristic' - including, but not limited to;

- Suspicious network activity
- Suspicious process activity
- Suspicious file activity
- Abnormal resource usage
- Abnormal user experience
  - Popups
  - Having to enable Word macros
  - Unusual file extensions
  - etc.

These additional factors may be present even in techniques that bypass Windows Defender, so it is important that we not only measure how well a program is able to run undetected, but also its *'stealth'* and how suspicious it may appear to a user who is executing the program.

For this purpose, we may evaluate how suspicious a program 'looks' heuristically, and take that into account as well as the score under our evaluation metric when determining how effective a technique may be.

Once we have determined the effectiveness of our range of basic techniques, we shall perform an ablation study of each technique to determine which aspects of its implementation cause it to be (or to not be) detected. In most cases, this will involve taking out the malicious payload from the technique (making it effectively useless), and seeing if the techniques' score improves.
Ablating each technique, will in short, allow us to extract exactly which lines of code are causing a program to be considered malicious by Windows Defender.
We may then analyse the results from this experimentation, and use it to guide our design of a novel technique with the goal of minimising detection (as defined by our evaluation metric) and maximising stealth (as defined by our 'suspicion' heuristic).

IV. Experimental Results

To summarise the results of the literature review, we present a table containing several techniques we have tested under our framework, as well as their score based on the aforementioned evaluation metric.

	Technique				
Metric	Obfuscation	Process Injection	Shellcode Execution	DLL Injection	Crypter
Defender Static	0	0	0	0	0
Defender Dynamic	0	0	0	0	0
VirusTotal	$\frac{31}{72}$	$\frac{38}{72}$	$\frac{40}{72}$	$\frac{26}{72}$	$\frac{40}{72}$
Score (1 d.p)	**11.4%**	**9.4%**	**8.8%**	**12.8%**	**8.8%**

TABLE II
Experimental Results

We implemented each technique in C, according to both the originating paper, as well as their commonly accepted implementation. [1]

As we can see, none of the techniques we studied are particularily promising on their own - all are detected by Windows Defender both on disk and dynamically, and none have particularly good VirusTotal scores.

It is important to note that each technique contains a sample malicious payload - in the form of shellcode, or a dll where applicable - that will perform some potentially malicious (but in this case, harmless) action, like opening an application or creating a file.

Some techniques, such as Process Injection and DLL injection still offer desirable stealth capabilities that will satisfy our heuristic, but that is no good if they are detected by Defender. Despite this, we will continue by discussing each technique in depth, with our ablative method to aid in our analyses, with the hopes of determining ***why*** these techniques don't work in order to improve them.

V. Ablative Study

Obfuscation

Obfuscation is somewhat of an interesting technique to nail down. There are many different methods one can use in order to achieve the obfuscation of a malicious program, including, but not limited to [15];

- Obfuscating malicious strings.
- Changing control structures or execution flow.
- Adding 'junk code' to change the file signature.
- Reordering malicious code to change the file signature.
- Hiding malicious code snippets throughout legitimate code.

For this reason, we decided to choose a standard implementation that was already available in tool format which we identified from our literature review; PeCloak.

PeCloak takes an executable file, performs some obfuscation on it (including rearranging, hiding malicious strings, and adding junk code), and outputs an 'obfuscated' executable [16].

[1]Source code for these implementations, as well as the ablated versions can be found at https://github.com/noahostle/BypassTechniques

We tested this originally with a Defender flagged 'malicious' executable, as well as an 'ablated' executable which does nothing at all and is not flagged, the results are as follows;

Metric	**Original Technique**	**Ablated Technique**
Defender Static	0	0
Defender Dynamic	0	0
VirusTotal	$\frac{31}{72}$	$\frac{12}{72}$
Score (1 d.p)	**11.4%**	**16.7%**

TABLE III
OBFUSCATION ABLATION STUDY

We see that this tool actually does more harm than good against current versions of Defender. Even the ablated executable was considered malicious despite containing only the obfuscations created by PeCloak when before being put through the tool, it was obviously not detected. We believe that this is because Defender has been configured to recognise executables which have been obfuscated by this tool.

We concede perhaps that manual obfuscation or a new update to the methods used by the tool could prove more effective, but we believe that this is not worth pursuing as newer versions of Defender will come to signature this method, just as they have for PeCloak.

Shellcode Execution

Shellcode Execution is a very simple technique wherein we simply allocate some executable memory in our own process, then place our shellcode there and direct the execution flow of our program to this memory [13].

For the shellcode execution technique, we chose to ablate the malicious payload out from the source file. Where there was previously malicious shellcode stored within the file as an array of bytes, we replaced this array with a single NOP (No Operation) byte, 0x90. This makes the technique ***completely non-functional*** without further modification, as it will do nothing when it is run.

This will however, allow us to determine wether Windows Defender is detecting the technique as malicious, or just the shellcode it is executing.

We retest this ablated sample and obtain the following result;

Metric	**Original Technique**	**Ablated Technique**
Defender Static	0	1
Defender Dynamic	0	1
VirusTotal	$\frac{40}{72}$	$\frac{17}{72}$
Score (1 d.p)	**8.9%**	**95.3%**

TABLE IV
SHELLCODE EXECUTION ABLATION STUDY

This result is very interesting, since removing the payload results in a much higher score, it seems to suggest that Windows Defender only detects the payload as malicious, and not the code for the technique itself. In other words, in the absence of a malicious payload stored within the file, the shellcode execution technique is not detected by Defender. Therefore, we have found a way to execute shellcode so long as the shellcode we execute is not stored within the file.

This is a very desirable property in terms of our metric (being detected by Defender), but since executables using this technique simply run the shellcode within their own memory, this on its own does not satisfy our stealth heuristic very well, apart from the fact that *it does not require the payload to be stored in an additional file on disk.*

Process Injection

Process Injection is a technique where we take some shellcode we wish to execute, then allocate some executable memory within the target process, and place our payload there, where we can then instruct Win32API to begin execution at the top of our payload. This will cause our payload to be running from inside the target process, and it will appear as if the target process if performing all of our malicious actions [20].

For Process Injection, we again chose to perform our ablation study by removing the malicious payload and measuring the effects this has on the payloads detectability. Just as we did for Shellcode Execution, we replace the payload with a **NOP** byte, this makes the technique ***completely non-functional***, but allows us to get a better insight into how Defender might be scanning for this technique.

Metric	Original Technique	Ablated Technique
Defender Static	0	0
Defender Dynamic	0	0
VirusTotal	$\frac{38}{72}$	$\frac{17}{72}$
Score (1 d.p)	**9.4%**	**15.3%**

TABLE V
PROCESS INJECTION ABLATION STUDY

Interestingly, as opposed to shellcode execution, we see that process injection is still detected by both Windows Defenders' dynamic and static analysis. This indicates to us that this technique is not being picked up because the payload it uses is signatured by Defender, rather that Windows Defender can detect the instructions that make up the source code for the technique itself, even when it is just on disk.

This is unfortunate, since as we have seen in the literature review, process injection has many very desirable characteristics in terms of satistfying our 'stealth' heuristic, including;

- Hiding suspicious process activity within a legitimate process - Curious users who open task manager will only see a verified Windows Process (Eg. Explorer.exe).
- Hiding network activity in a legitimate process - Curious Users will only see whatever process we inject into as the source of network traffic, which they will be more likely to trust to communicate on their network.
- This technique does not need an additional file stored on disk, and once a payload is injected, the original parent process may be halted, and the malicious executable can be altered or removed, leaving little trace of the malware except that which can only be found using forensic techniques.
- This technique does not cause any abnormal resource usage, or user experience issues that would be evident to the a User of average computing knowledge.

This gives us a challenge, as finding a way to smuggle these characteristics past Defender would be very beneficial for the stealth of our final payload, but as is evident by the results of this experiment, we cannot hope to just compile this technique to an executable and run it off disk.

DLL Injection

DLL Injection is a bit of a tricky case as it requires two separate files in order for the technique to work. A common implementation of DLL injection involves allocating some memory within the target process, then storing the *filepath* of the malicious .dll, then invoking the Win32API loader *'LoadLibrary'* on the memory address that points to the filepath [26]. The Windows loader will then find our dll at the path specified, and attach it to the process whose memory the path is stored in.

For this technique, we can use a single 'Injector' executable, whose job it is to allocate and inject a filepath into a remote process, then invoke the LoadLibrary, and two different .dll payloads - One for the original technique (that invokes a malicious command), and one for the ablated technique (that is completely empty other than the necessary structure of a dll).

Note that the VirusTotal score in this case is the amount of unique AV's that detected the Injector.exe, and the Payload.dll combined.

Metric	Original Technique	Ablated Technique
Defender Static	0	0
Defender Dynamic	0	0
VirusTotal	$\frac{26}{72}$	$\frac{25}{72}$
Score (1 d.p)	**12.8%**	**13.1%**

TABLE VI
DLL INJECTION ABLATION STUDY

From the data we see that much like process injection, even ablating the malicious payload out does not make DLL injection undetectable. There is one caveat however, using the original technique, we find that Defender detects both the injector executable, as well as the malicious payload dll as malicious. However, when the malicious contents of the DLL are ablated out, and only the core structure is left, the DLL is no longer detected, but the injector executable still is. This is not useful however, as the injector executable is necessary for the execution of the DLL, and the executable is considered malicious by Defender regardless of ablation.

From a heuristic perspective, this technique is similar to process injection in that it hides malicious activity in a legitimate process, but with a large drawback in that a separate .dll file must be downloaded in addition to the main executable, and stored on disk. This increases the chances of detection via static analysis and also could raise user suspicion when they see an additional file with a strange extension, neither of which are desirable characteristics for our evaluation metric or our stealth heuristic.

Crypter

A crypter is a type of software that can encrypt, obfuscate, and manipulate malware, to make it harder to detect by security programs. It is used by cybercriminals to create malware that can bypass security programs by presenting itself as a harmless program until it gets installed.

A crypter contains a crypter stub, or a code used to encrypt and decrypt malicious code. Here, the encryption method was a simple XOR of the file and some key. In practice, more complex encryption methods would be used to further evade Windows Defender. Here, 3 files are necessary on the threat actor's side [4]:

- The malware to be encrypted (e.g. keyboard logger, network traffic logger, etc.)
- The main crypter file, to perform the encryption on the chosen malware
- The stub, which, when executed, will decrypt the malware and run it

On the user's side, they must install both the encrypted malware file, and install and run the stub. In practice, that can often be combined into one file, but for the purposes of demonstration it is not necessary. Getting the file onto the user's computer can be achieved through various methods, but will commonly involve some form of social engineering, tricking a user to click on a malicious link and download the program.

The original technique used a keyboard logger made from scratch that uses Win32 API calls to log key presses to an output file. This, naturally, did not perform well against Windows Defender or on VirusTotal (scores can be seen in the table below).

For the ablative study, the "malware" encrypted was a one-line program that echoes Hello World to the console. When downloaded into the virtual environment, Windows Defender was able to immediately recognise and quarantine the program. Many other anti-viruses are also easily able to flag the stub as a Trojan when it was tested on VirusTotal. This is likely due to the fairly trivial signature a crypter has – Windows Defender could detect the program even when nothing malicious was being run. The results are summarised below.

Metric	**Original Technique**	**Ablated Technique**
Defender Static	0	0
Defender Dynamic	0	0
VirusTotal	$\frac{40}{72}$	$\frac{14}{72}$
Score (1 d.p)	**8.8%**	**16.1%**

TABLE VII
CRYPTER ABLATION STUDY

VI. Rationale of our Novel Architecture

After analysing the techniques we collected through our ablation study, we arrived at a few criteria that we thought would be necessary for a good design.

Firstly, we should try to move our malicious payload off disk if possible, as techniques which rely on a malicious payload stored in a file seemed to perform worse overall - especially during static analysis - compared to those that didn't.

Secondly, at least some component of our design should feature 'stealth' enhancing functionality, rather than just trying to noisily execute a payload so long as it is not flagged by Defender - it is argued that all of the antivirus bypass methods in the world mean nothing if upon your payload executing the user is immediately made aware of the suspicious activity and decides to investigate or have their computer checked by a professional.

Finally, and most importantly, we must employ techniques which by clever design (and based on the analyses from our experiments) seem as close to a legitimate program in Defenders eyes as possible in order to evade detection (this will be elaborated on later).

With this criterion in mind, we decided upon the following construction;

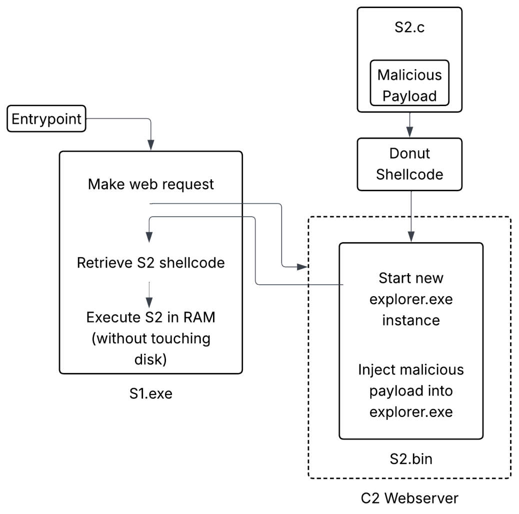
S2.c
Malicious Payload
Entrypoint
Donut Shellcode
Make web request
Retrieve S2 shellcode
Execute S2 in RAM (without touching disk)
S1.exe
Start new explorer.exe instance
Inject malicious payload into explorer.exe
S2.bin
C2 Webserver

Fig. 1. Architecture of our Proposed Method

In this design, we take advantage of several features of the techniques that we experimented with in order to satisfy our criterion.

We shall start with the entrypoint of our technique - S1.exe, based on the *Shellcode Execution* technique. As noted by our experimentation, shellcode execution is easily detected by defender, but is not detected when the malicious payload is stripped from the file. To solve this issue we store the malicious payload externally (for example, on a web server), and only retrieve it at run time, thus completely evading static detection.

We find by testing, that dynamic detection is also evaded allowing our payload to execute entirely undetected by Defender. This fairs greatly for our evaluation metric (since Defender is bypassed) but not so greatly for our stealth heuristic. As aforementioned, the shellcode execution technique runs the malicious payload in its own memory. This means that for the entire execution time of our malicious payload, the user will see a console window, as well as a suspicious looking process in task manager - which they may force end - and will likely become aware of our malicious actions. This can be particularly impactful if our final payload is a reverse shell, or other similar remote access implant (or indeed any final malicious process which must remain alive for some time in order to achieve its goal), as discovering and ending the process would cut our access short.

In order to solve this issue, we would like to employ some technique that has a good stealth heuristic. For this purpose, we have chosen process injection, since as per our previous analyses, it provides more stealth to the process than any average user would be able to uncover by hiding our payload inside a legitimate process. This makes any final payload appear as the legitimate application we inject into (take for example, Windows File Explorer) when appearing in task manager, network logs, Windows security audit logs, etc. Therefore we conject that the average user would be very unlikely to investigate and end - or realistically even notice - our malicious process before it has accomplished its goal.

The issue with doing this however, is that of the techniques we tested in our experiments, all the ones we evaluated to have good stealth heuristics (dll injection and process injection) were detected quite readily by Windows Defender, even when we removed the payload.

To reconcile this issue, we employ an open source tool called *Donut*. Donut is a tool written by a team led by *'TheWover'* that can take a Windows Executable and convert it into shellcode. This is necessary because the shellcode execution technique is only capable of executing what is known as *'Position Independant Shellcode*. This is because 'shellcode' in this context refers to an array of assembly instructions, and when we execute it using this technique, we are essentially just allocating some executable memory space for this array, then pointing the program counter to the start of it, and letting it execute on the CPU. When we compile a C source file, it is compiled into a *Position **Dependent*** executable, which is built in such a way that it expects to be able to use the Windows Loader to resolve dependencies and map itself into memory. If we try to make the CPU execute this instruction by instruction, it will not work as without the loader the CPU is not privvy to the relative memory locations of certain things our payload needs to run. To resolve this, Donut is capable of taking an executable, and wrapping it in its own *position **independant*** version of the Windows Loader. This allows the resulting binary output to be executed as shellcode [22].
Coming back to our design, we can leverage this ability to compile any arbitrary C source file to shellcode in order to execute the process injection technique in a way that is undetected by Defender. We do this by writing some C code to perform process injection (S2.c) and compiling this for our target machine and running it through Donut, producing the shellcode 'S2.bin'. S2.bin can then be served over the internet to an instance of S1.exe, where it will be executed. This exploits shellcode execution's negative detection status when retrieving its shellcode over the internet, and drastically improves it's stealth heuristic by moving the delivery of the final malicious payload to process injection. It is crucial to recognise that these techniques on their own were not considered effective; shellcode execution lacked stealth heuristic, and process injection failed to bypass Defender. However when combined under our design (with help from Donut) these techniques create a payload delivery system which satisfies our criteria on both accounts.

To examine how this works more deeply we may look at the structure of each components memory as the flow of execution progresses through our design.

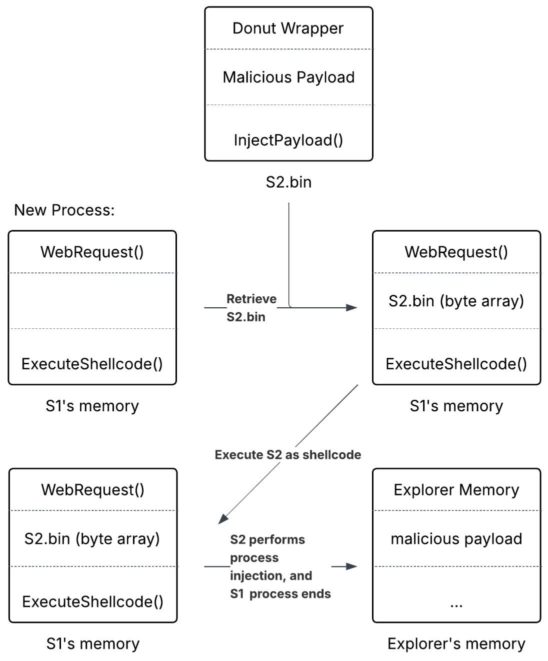

Fig. 2. Memory regions throughout execution

Firstly, the user will download S1.exe to their disk, which based on our experimentation, Defender will not detect, as the shellcode is absent.

Then, S1 will reach out to the attackers' server, downloading S2.bin - our process injection technique compiled using Donut.

Once S1 has S2 in its memory as an array of bytes, it will allocate some executable memory within its own memory space (inside the S1.exe process) and place S2.bin into it, and instruct the CPU to begin executing from that memory location as shellcode.

Thanks to Donuts compilation, this will execute S2 as if it were a compiled .exe file without it ever having to touch the disk - entirely avoiding static analysis.

S2, while running from S1's memory region, will then invoke an instance of Windows Explorer, and inject the final malicious payload into it. Once this process is complete, S1 has finished its task, and the S1 process, along with the array that contains S2 can be killed and removed from RAM. This means that S2 never exists on disk, and only exists in RAM for the tiny amount of time it spends executing, then any trace of it, along with S1's suspicious looking process (which on its own Defender does not detect anyway) will end.

Finally, a new instance of Explorer.exe will be left containing our malicious payload, and any other traces of S2.bin, or S1's process will be gone, leaving our malicious payload to run for however long it wants masquerading as a signed Windows process

VII. RESULTS

Metric	Our Design
Defender Static	1
Defender Dynamic	1
VirusTotal	$\frac{14}{72}$
Score (1 d.p)	**96.1%**

TABLE VIII
EVALUATION OF OUR DESIGN

It is important to note that this score is while using a full malicious payload, and also that a very good stealth heuristic is achieved due to our design.

A source code repository, including an automated bash script that takes in a final malicious payload and sets up the whole environment automatically, as well as a demonstration video can be found linked in the Footnotes. [2]

[2] Source Code Repository and Demonstration Video found at: https://github.com/noahostle/DefenderBypass

VIII. ANALYSES AND CONCLUSION

Based on the results, we can construct a number of attacks, the most interesting of which may be altering the icon and name of our S1 executable in order to spoof a legitimate signed Windows Process, however we can also draw several impactful conclusions on the nature of Defender.

To start, it is important to note that in our testing we used a final malicious payload generated by a tool named ***msfvenom***. Msfvenom is a tool widely used in the security community because it makes generating shellcode for certain simple tasks (such as a reverse shell) much easier. Because it is so widely used however, all of its payloads are **heavily** signatured by Windows Defender. This evidence is highly anecdotal, but ask any malware author or analyst and you will receive the same notion in response - that msfvenom payloads are frequently and easily detected by Windows Defender. We even use a payload generated by this tool in our testing of each technique. This idea is supported by the fact that shellcode execution is not detected without the msfvenom payload, but immediately detected with it.

Therefore, we say that it is of note that when using the exact same msfvenom payload in combination with our technique (with no obfuscation or other trickery) that Defender does not detect it.

Much analysis remains to be done on the precise method by which Defender detects certain payloads (and we suspect this analysis would be done with great difficulty due to Defender's being closed source), but we conject that the fact that such a highly signatured payload remains undetected when delivered using our technique suggests that Defender does not scan or monitor the right locations in order to protect against such an attack - that this might not be a matter of Defenders threat database being a bit behind the next wave of malicious techniques, but rather that this technique evades the places Defender is looking in entirely. What adds strength to this thought is that we are not developing any new base techniques whose signatures or behaviour merely have to be memorised, then all similar attacks can be blocked. Instead, we are using techniques only that have been known and defended against for a long time, but arranging them in such a way that Defender as it stands *cannot* detect them without employing new specific methods. Strictly speaking, if Microsoft's threat model includes that;

1) The act of creating a new function from an array of bytes (shellcode execution) is not necessarily malicious.
2) Defender does not need to deeply inspect a processes RAM for malicious payloads as it is running.

Then it logically follows that under this threat model our technique **cannot** be detected.

We suspect that this may be the case, but even if it is not, and we have only succeeded in stringing together techniques in a way that once detected, can be signatured and blacklisted, we have at the very least shown that a skilled attacker can realistically create a new strain of malware that will work against all current versions of Defender until their victims install Microsoft's next patch. In this weakest case, we find it concerning that a single undergraduate student with no funding or organisational/nation-state motivation is capable of defeating Defender in this way.

To conclude, we see that through our experimentation with common techniques, our analyses, and the efficacy of our novel design, we were able to prove our hypothesis; *'Windows Defender's focus on signature-based analysis leaves it vulnerable to bypass tactics, allowing obfuscated malware to execute without triggering alerts.'* to be true.

We have shown that it is possible to rearrange and modify well known techniques in a way that potentially evades Defenders' current scanning paradigm, and at the very least bypasses its signature based analysis. We have demonstrated that our technique can bypass Defender in order to retrieve and execute arbitrary malicious code - no matter how singatured - and in doing so, we have also shown that a lone skilled attacker is capable of devising a method to defeat Defender.

Based on these results, we come to the unfortunate conclusion that the latest version of Defender cannot be considered a sufficient defense for users amidst a modern threat landscape, and that any user who relies on Windows Defender for their sole protection against malware is at risk of undetected compromise.

References

[1] A. Garba, F. (n.d.). “Evaluating Antivirus Evasion Tools Against Bitdefender Antivirus”. Fintech. https://www.researchgate.net/profile/Aliyu-Musa-6/publication/365438254_Evaluating_Antivirus_Evasion_Tools_Against_Bitdefender_Antivirus/links/6374f6e854eb5f547cda02e8/Evaluating-Antivirus-Evasion-Tools-Against-Bitdefender-Antivirus.pdf

[2] Al-Awadi, Y. M., Baydoun, A., & Rehman, H. U. (2024). “Can Windows 11 Stop Well-Known Ransomware variants? An examination of its built-in security features”. Applied Sciences, 14(8), 3520. https://doi.org/10.3390/app14083520

[3] Ali, Y., Hameed, A. ”Shibboleth Authentication Request.” (2025). IEEEXplore https://ieeexplore-ieee-org.ezproxy.lib.uts.edu.au/document/8994615

[4] Ali, Y., & Hameed, A. (2019). “Cloud Crypter for bypassing Antivirus”. IEEE, 1–6. https://doi.org/10.1109/icet48972.2019.8994615

[5] Alvarado, A., Portillo, S. ”Understanding Crypter-as-a-Service in a popular underground marketplace.” (2020). Arxiv.org. https://arxiv.org/html/2405.11876v2

[6] Aminu, S. A., Sufyanu, Z., Sani, T., & Idris, A. (2020). “Evaluating the effectiveness of antivirus evasion tools against windows platform”. FUDMA Journal of Sciences, 4(1), 112–119. https://www.fjs.fudutsinma.edu.ng/index.php/fjs/article/download/27/17

[7] ”Antivirus and EDR Bypass Techniques Explained.” (2024). VAADATA. https://www.vaadata.com/blog/antivirus-and-edr-bypass-techniques/

[8] Blaauwendraad, B., & Ouddeken, T. (n.d.). “Using Mimikatz’ driver, Mimidrv, to disable Windows Defender in Windows”. Os3.nl. https://www.os3.nl/_media/2019-2020/courses/rp1/p61_report.pdf

[9] Chatzoglou, E., Karopoulos, G., Kambourakis, G., & Tsiatsikas, Z. (2023). “Bypassing antivirus detection: old-school malware, new tricks”. Proceedings of the 17th International Conference on Availability, Reliability and Security, 1–10. https://doi.org/10.1145/3600160.3605010

[10] Delaney, J. (n.d.). “The Effectiveness of Antivirus Software”. ProQuest. https://www.proquest.com/openview/d3ff27e1e773c8dd36e7746e64567702/1.pdf

[11] Dimov, R., & Savova, Z. (2024). “Antivirus performance evaluation against powershell obfuscated malware”. Environment Technology Resources Proceedings of the International Scientific and Practical Conference, 4, 71–78. https://doi.org/10.17770/etr2024vol4.8201

[12] Ivanov, I., Dimitrova, M., & Atanasova, M. (2023). “Antivirus bypass of payloads to launch advanced client-side attacks”. IEEE, 1–3. https://doi.org/10.1109/telecom59629.2023.10409661

[13] Johnson, A., & Haddad, R. J. (2021). “Evading Signature-Based antivirus software using custom reverse Shell exploit”. SoutheastCon, 1–6. https://doi.org/10.1109/southeastcon45413.2021.9401881

[14] Kaushik, K., Sandhu, H. S., Gupta, N. K., Sharma, N., & Tanwar, R. (2022). “A Systematic approach for evading antiviruses using malware obfuscation”. In Lecture notes in electrical engineering (pp. 29–37). https://doi.org/10.1007/978-981-16-8774-7_3

[15] C. Kalogranis, ”AntiVirus Software Evasion: An Evaluation of the AV Evasion Tools.” Ph.D. Thesis, Piraeus, Greece,: University of Piraeus, Department of Digital Systems, 2018. https://dione.lib.unipi.gr/xmlui/handle/unipi/11232

[16] M. Czumak, “peCloak.py - An Experiment in AV Evasion - Security Sift,” Security Sift, Mar. 09, 2015. https://www.securitysift.com/pecloak-py-an-experiment-in-av-evasion/

[17] Mohamed, N. (n.d.). “Study of bypassing Microsoft Windows Security using the MITRE CALDERA Framework”. F1000. https://f1000research.com/articles/11-422/v3

[18] Parveen, N. K. (2024). “Advanced techniques of malware evasion and bypass in the age of antivirus”. International Journal for Electronic Crime Investigation, 8(3). https://doi.org/10.54692/ijeci.2024.0803200

[19] Pham, H., Nguyen, V. T., Tiep, M. V., Hien, V. T., Huy, P. P., & Vuong, P. T. (2021). “Evading security products for credential dumping through exploiting vulnerable driver in Windows operating systems”. In Communications in computer and information science (pp. 486–495). https://doi.org/10.1007/978-981-16-8062-5_36

[20] Shawwal Adam, A. (n.d.). “Survey of EDR Evasion Techniques, Trends, and Taxonomy for Classifying Modern Attacks”. ProQuest. https://www.proquest.com/docview/3151978918/fulltextPDF/6BC72B406D96498DPQ/1

[21] Szeto, N. (2024). Kushida Keylogger: “A study on malware Evasion techniques. SSRN Electronic Journal”. https://doi.org/10.2139/ssrn.4668226

[22] TheWover, “TheWover/donut: Generates x86, x64, or AMD64+x86 position-independent shellcode that loads .NET Assemblies, PE files, and other Windows payloads from memory and runs them with parameters,” GitHub, Oct. 23, 2024. https://github.com/TheWover/donut/

[23] Vasani, V., Bairwa, A. K., Joshi, S., Pljonkin, A., Kaur, M., & Amoon, M. (2023). “Comprehensive analysis of advanced techniques and vital tools for detecting malware intrusion”. Electronics, 12(20), 4299. https://doi.org/10.3390/electronics12204299

[24] ”What is Crypter? - The Evolving Tactics of Malware Obfuscation”. (2023). Reasonlabs.com. https://cyberpedia.reasonlabs.com/EN/crypter.html

[25] Tibirna, L. ”The Architects of Evasion: a Crypters Threat Landscape”. Sekoia.io Blog. https://blog.sekoia.io/the-architects-of-evasion-a-crypters-threat-landscape/

[26] Yehoshua, N., & Kosayev, U. (2021). “Antivirus Bypass Techniques”. Packt. https://books.google.com.au/books?hl=en&lr=&id=Gpw3EAAAQBAJ

[27] Yuceel, H. ”MITRE ATT&CK T1055.001 Process Injection: DLL Injection.” (2025) Picus Security. https://www.picussecurity.com/resource/blog/t1055-001-dll-injection