Malware Obfuscation: Various techniques

Nabeel Sheikh
CSCE 451

Abstract

Anti-malware products are an ubiquitous and well researched defense mechanism in the industry, yet thousands of new malware variants are able to evade its security.

The arms-race between anti-malware software and malware authors is a continuing battle. The increasing complexity of malware today has called for a real need to develop proper tools to catch the increasing complexity of malware. The availability of knowledge for aspiring malware creators has caused a massive surge of malware, be it naive or incredibly innovating.

Obfuscation, by various techniques, is widely used by malware to evade these antimalware scanners. In this paper, we will explore techniques such as malware encryption, polymorphic and metamorphic malware, rootkits, process injection, and packers.

1 Encrypted Malware

The typical approach is to encrypt the main body of the virus and then decrypt/recover the body when the infected file is to run. Each infection carried a different key making the encryption unique and thus immune to signature searching by anti-malware scanners. The first piece of malware to use encryption to scramble contents was the Cascade virus, which was in 1986. This virus consisted of an encryption/decryption routine followed by the body of encrypted code(eventually adopted idea by most encrypted malware). Cascade used a symmetrical XOR cipher(figure 1) with the key based on the file size. Even though this is a weak cipher by today's standards, it was very effective back then. Anti-virus software at the time was based on simple pattern matching and had a very difficult time with pattern matching. With the body of the malware encrypted at infection time, the only part of code that could be fingerprint-able was the XOR encryption/decryption routine which was static. The anti-virus software could not distinguish between different mutations of the same malware that shared the same routine. Also, as the size of these malwares decreased, the number of false positives by AV programs increased. Another thing to note, the XOR operation is symmetrical and could be done with a single function for both encryption and decryption. [1]

Plain Text	1	1	0	1	0	0	1	1
Key (Apply)	0	1	0	1	0	1	0	1
XOR (Cipher Text)	1	0	0	0	0	1	1	0
Key (Re-Apply)	0	1	0	1	0	1	0	1
XOR (Plain Text)	1	1	0	1	0	0	1	1

Figure 1: XOR Cipher

As time went on, anti-malware products were able to properly match most decryptor patterns with growing amounts of encrypted malware. This eventually led to more complex encryption methods to help combat the static encryption/decryption flaw.

1.1 Polymorphic Malware

Polymorphic malware repeatedly packs and encrypts to change its appearance. The malware has the ability to change every time it changes its location or is prompted remotely to change. As a polymorphic malware infect file to file, it would radically change how it would conceal itself while maintaining its functionality (Figure 2). With a very complex and properly implemented polymorphic virus, there should be no pattern in the decryption code thus leaving no signature for an AV to scan. With the Mutation Engine, malware writers were able to use their non-obfuscated code and turn it into polymorphic malware with ease. In order to defeat this, emulation was used to "sandbox" the untrusted programs. This would allow the execution of the program in a walled-off environment where the suspected software could not do any harm to the system components. During runtime, the AV would scan the suspect program's memory footprint against others and flag suspicious behavior such as modification to other executables or writes to the drives boot sector. Malware writers responded by armoring their malware with various techniques mostly exhaustive or too power hungry for early emulators (e.g endless looping, floating point operations, fringe features). However, this would not last long as computing power grew and emulation matured.[1]

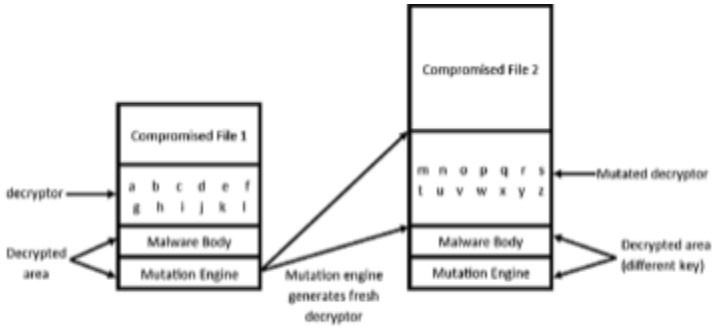

Figure 2: Polymorphic Malware

2 Rootkits

Rootkits are interesting because they allow malware to bury itself deep into the system and hide from the OS. Modern machines have differing privilege levels of executions or rings (Figure 2). An attacker's malware can trick a user level program by essentially "lying" to it. For example, if a user program calls for a file scan, the rootkit can trick/lie. It can easily pass over the desired files and return the wrong results. Due to virtualization, some rootkits actually have higher privileges than the kernel. How does this help obscure itself? It can hide itself from file scans, network connections, and the OS. Types of rootkits include - kernel, hardware/firmware, and hypervisor/virtualized rootkit. [10]

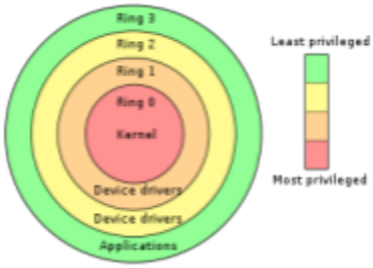

Figure 3: OS Privilege rings

Overall, a rootkit is a tool that is made to hide itself from other processes, data, and activity on a system.[2]

2.1 How do they spread?

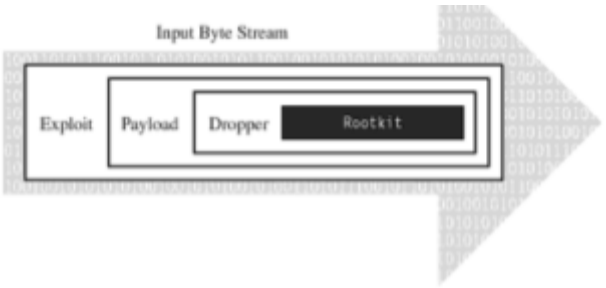

Figure 4: Rootkit Deployment

Rootkits usually consist of multiple stages - The exploit, and the payload consisting of the Dropper and rootkit(Figure 3). A dropper is there to help make it past the initial security scan,transform the rootkit(e.g compress, encode, encrypt) and encapsulate internally. When the dropper is then executed by a user, the rootkit will be unpacked and decrypted. The dropper should delete itself leaving only the rootkit and its dependencies. Many droppers will not even include the rootkit as part of the payload but instead download the rootkit from a remote location. In some cases, it may even download a second program which then downloads the rootkit. The reason for this is to minimize the evidence left behind by the dropper. For example, if the dropper fails to delete itself, a forensic investigator will not be able to do any analysis on the rootkit code itself.[3]

Other means include social engineering to get console access(e.g. USB thumb drive, FTP/wget download).

2.2 Rootkits - Developing rootkits on Linux

We can create a rootkit in linux by developing a driver/ LKM(loadable kernel modules). This example will not include real malicious behavior but lay the groundwork on how to develop modern rootkits for linux.
Below is the skeleton code for the rootkit:

```
#include <linux/module.h>
#include <linux/init.h>
#include <linux/kernel.h>

MODULE_LICENSE("GPL");
int rooty_init(void);
void rooty_exit(void);
module_init(rooty_init);
module_exit(rooty_exit);

int rooty_init(void) {
 printk("rooty: module loaded\n");
 return 0;
}

void rooty_exit(void) {
 printk("rooty: module removed\n");
}
```

Next, we need to make the kernel object file with the makefile below:

```
Obj-m := rooty.o
KERNEL_DIR = /lib/modules/$(shell uname -r)/build
PWD = $(shell PWD)
all:
 $(MAKE) -C $(KERNEL_DIR) SUBDIRS=$(PWD)
clean:
 rm -rf *.o *.ko *.symvers *.mod.* *.order
```

Next, we need to load this module into our linux kernel. To do this, call `insmod rooty.ko`. With this our rootkit should be installed onto the linux kernel but we need to make sure to hide our module from `lsmod`. To do this we need to remove the entry from the /proc/modules the /sys/modules locations:

```
int rooty_init(void) {
 /* Do kernel module hiding*/
 list_del_init(&__this_module.list);
 kobject_del(&THIS_MODULE->mkobj.kobj);
```

Our rootkit should be installed and hidden from the common lsmod command. From here, we can use our rootkit to hook to a system call to hide the activities of the process.[4]

3 Process Injection

This technique allows a program to run code under another process. By doing this, malware creators can inject into constant Windows processes like notepad.exe making anti-virus software fail to find the malicious process. Malware can also hook to processes with networking capabilities to send and mask its own malicious packets. It involves running custom code within the address space of another process. We will look at some example techniques of process injection.

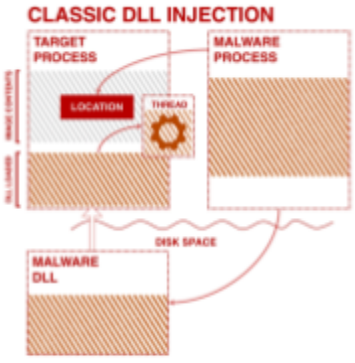

Figure 5: DLL Injection

DLL Injection is the most common technique used to inject malware into another process. The malware will load the path to its malicious DLL into the virtual address space of another trusted user process. The program will then call the malicious DLL and load the malware code into the memory space of the target process(Figure 5).

Portable executable injection is simpler in that instead of passing the address, the malware directly copies the malicious code into the target process. The malware will then cause it to execute by either shell code or calling the Windows API - CreateRemoteThread. PE injection allows the malware to not have to put the malicious code on the disk. The malware will allocate memory(VirtualAllocEx) and then write to the target process(WriteProcessMemory).

Process hollowing is when malware unmaps the target processes code from memory and overwrites it with malicious executable code. The malware will spawn the targeted process in suspended mode. Next, the malware will swap the memory content by unmapping the memory of the target process by calling ZwUnmapViewOfSection/NtUnmapViewOfSection. The malware will then allocate the new memory and use WriteProcessMemory to write the malware payload to the target process. Lastly, the malware process will resume the suspended target process.

Thread execution hijacking works similar to the other methods talked about. The malware will get the handle of the target thread, suspend it, and then perform an injection.

Asynchronous Procedure Calls(APC) injection is when malware forces another thread to execute the custom malicious code by attaching to the APC Queue of the target thread. Every thread has a queue of APCs; the malware will wait for a thread that is in an alterable state so that it can queue an APC to that thread. One notorious example is AtomBombing, which relies on APC injection. Moreover, it used the atom table in modern Windows systems. Atom Bombing consists of a malicious process creating a global atom and storing it in the global atom table. It will then use APC to make the target process run the function to get the malicious atom from the global atom table. The target process will then call the malicious function and copy the malicious machine code into its process memory space(Figure 6). [5]

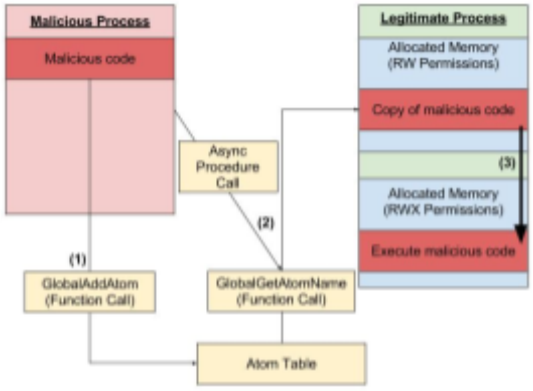

Figure 6: AtomBombing Process

Atom Bombing Process:

1. Split the payload into NUL-terminated strings
2. Create an Atom for each one (GlobalAddAtom). Note: Atom cannot represent a 0-length string c.
3. Copy the strings to the target process memory using NtQueueApcThread(thread, GlobalGetAtomName, atom,target_address,size).

Example code snippet from Amit Klein and Itzik Kotler's, 2019 Windows code injection survey[9]:

```
HANDLE th = OpenThread(THREAD_SET_CONTEXT| THREAD_QUERY_INFORMATION, FALSE,
    thread_id);
ATOM aux = GlobalAddAtomA("b"); // arbitrary one char string
if (target_payload[0] == '\0')
{
    printf("Invalid payload (starts with NUL)\n");
    exit(0);
}
for (DWORD64 pos = sizeof(target_payload) - 1; pos > 0; pos--)
{
    if ((payload[pos] == '\0') && (payload[pos - 1] == '\0'))
    {
        ntdll!NtQueueApcThread(th, GlobalGetAtomNameA, (PVOID)aux,(PVOID)(((DWORD64)target_payload) + pos-1),
(PVOID)2);
    }
}
for (char* pos = payload; pos < (payload + sizeof(payload)); pos += strlen(pos)+1)
{
    if (*pos == '\0')
    {
        continue;
    }

    ATOM a = GlobalAddAtomA(pos);
    DWORD64 offset = pos - payload;
    ntdll!NtQueueApcThread(th, GlobalGetAtomNameA, (PVOID)a,
    (PVOID)(((DWORD64)target_payload) + offset), (PVOID)(strlen(pos)+1));
}
```

4 Metamorphism

Metamorphism is when the malware is able to reprogram itself as it moves generation to generation. The code is able to parse and mutate as it moves to new hosts. We can classify these metamorphic malware by analyzing these two attributes- Communication and Transformation[8].

4.1 Communication [8]

Open-World - Able to communicate with the outside world, that is, download necessary data or send data.

Closed-World - Makes no communication outside the host.

4.2 Transformation [8]

Binary Transformer - Binary mutates during evolution.

Alternative Representation Transformer - During evolution, will carry source-code and infect machines with a compatible compiler.

Register Swapping - Usually registers are designed for specific instructions. However, they can be used against conventions.

Code Substitution - Switch instructions for ones that look different but produce an equivalent result.

Branch Condition Reversing - Reordering branch conditionals with no effect to code behavior.

Garbage Insertion - Inserting dead code like nop and clc that change the binary but not the executables behavior.

Subroutine Reordering - Moving the order of the subroutines so that they appear to be called in random order.

Code Insertion - insert malware instructions in-between host instructions.

4.2 Detecting Metamorphic Malware

Metamorphic malware is very hard to pattern match since its signature is constantly changing every generation. This leaves AV scanners who entirely rely on pattern matching stumped. In order to detect and defeat metamorphic malware, AV scanners will need to develop advanced heuristics and other detection methods to effectively identify metamorphic malware. An interesting topic of discussion is using deep learning(LTSM) to correctly identify metamorphic malware.

5 Packers

Packers were very popular in the 80s and early 90s when RAM and disks were much smaller and executable sizes were given major consideration. Packing tools were designed to encrypt and compress files. Malware authors adopted these tools to add polymorphism, armoring, metamorphism, entry point obfuscation, etc to hide from AV software. Packers ensure that even a small insignificant change to the malware will completely change the signature of the packed malware(similar to a cryptographic hash function). Some popular packers are Polypack and Themida. Polypack was very popular because it allowed malware authors to test a multitude of packers and compare the effectiveness against many AV scanners. This allowed authors to quickly model what packers to use and systems to attack to achieve maximum efficiency. Themida was a packer known in the industry for its packing ability. Polypack showed that Themida outperformed all other packers and made use of many of the obfuscation techniques talked about in the metamorphism section. [1]

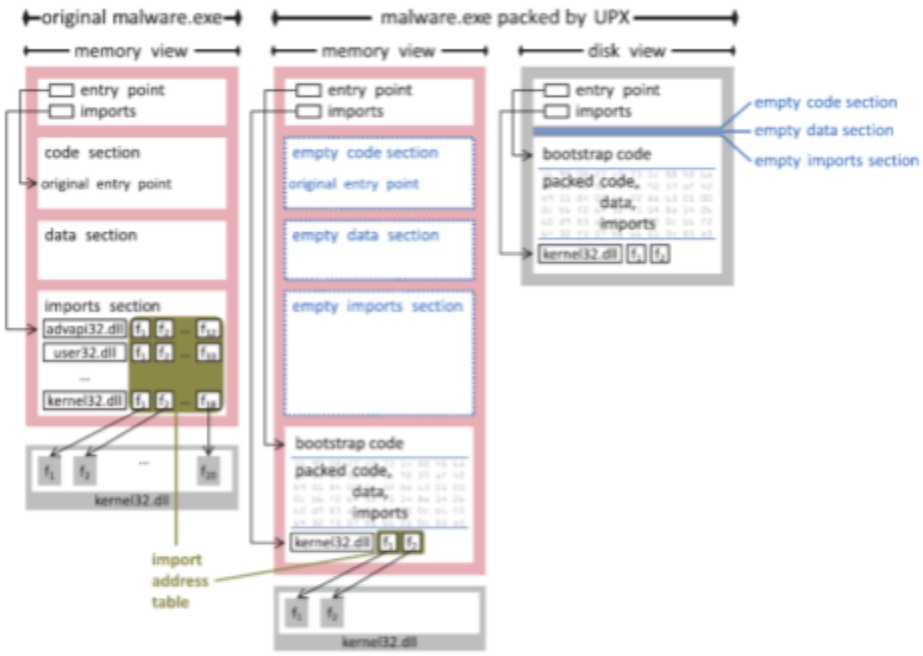

Figure 7: Packed Binary memory view

A packer tool will first set the executable entry point to the entry point of the bootstrap code that unpacks the malware code and moves control to the OEP. The packer will also pack the PE binary format's import table and import address table. This hides many significant details of the original executable. Sometimes static analysis can be done on the bootstrap code itself to reveal implementation details; to combat this, the bootstrap code itself can be packed. Onced packed, the malware executable, from the disk view, will have an empty code, data, and imports section making it much harder to statically analyze(Figure 7).[7]

6 Conclusion

The back and forth arms-race never ends but one can see that being on the defensive side always leaves you a step behind. It is much easier to develop a technique to obfuscate your code then have to find all possible ways a malware author can evade an AV scanner. However, this competition has brought many innovative and brilliant ideas to the computing industry that can be used for legal and practical usage and deepened the understanding for all those involved in the computing and security field.

References

[1] M. Schiffman, “A Brief History of Malware Obfuscation: Part 1 of 2 ”, http://blogs.cisco.com/security, Feb. 2010.

[2] Greg Hoglund, James Butler, “Rootkits: Subverting the Windows Kernel”, https://books.google.com/books?hl=en&lr=&id=fDxg1W3eT2gC&oi=fnd&pg=PR9&dq=what+are+rootkits+scholarly+articles&ots=e85_muIsoA&sig=YgCEjwyoeYmzBxX8PSEMIPiFHu4#v=onepage&q&f=false, 2006

[3] Bill Blundell, “The Rootkit Arsenal: Escape and Evasion in the Dark Corners of the System 2nd Edition”, 2013

[4] Tyler Borland, “Modern Linux Rootkits 101”, http://turbochaos.blogspot.com/2013/09/linux-rootkits-101-1-of-3.html, 2013

[5] Ashkan Hoesseini, “Ten process injection techniques: A technical survey of common and trending process injection techniques”, https://www.elastic.co/blog/ten-process-injection-techniques-technical-survey-common-and-trending-process, 2017

[6] Ilsun You and Kangbin Yim, “Malware Obfuscation Techniques: A Brief Survey”, https://profsandhu.com/cs5323_s18/yk_2010.pdf, 2010

[7] K.A. Roundy and B.P. Miller, “Binary Code Obfuscations in prevalent packer tools”, https://dl.acm.org/doi/pdf/10.1145/2522968.2522972, 2013

[8] Andrew Walenstein and Rachit Mathur and Mohamed R. Chouchane and Arun Lakhotia, “The Design Space of Metamorphic Malware”, 2007

[9] Amit Klein and Itzik Kotler, “Windows Process Injection in 2019”, https://i.blackhat.com/USA-19/Thursday/us-19-Kotler-Process-Injection-Techniques-Gotta-Catch-Them-All-wp.pdf, 2019

[10] Corey Nachreiner, “Malware Obfuscation: Various techniques”, https://www.darkreading.com/attacks-breaches/how-hackers-hide-their-malware-advanced-obfuscation/a/d-id/1329723, 2017