ANALYSIS OF WINDOWS PORTABLE EXECUTABLE SOFTWARE PROTECTION SYSTEMS

David Nachkebia

Executable Packers. Reverse engineering compiled binaries is itself a challenge. In the process that we can call "forward engineering", compiler generates the code that is not intended to be explored and comprehended (not to mention – modified) by the human being.Diversity of software and hardware platforms and corresponding compilers results in variety of binary formats and respective runtime execution environments;hence the reverse engineering process differs significantly for multitude of computing systems. Even for same targeted platforms different compilers produce binary compatible, yet slightly (and sometimes significantly) different code. But as long as the architecture of the compilation platform (binary format and instruction set) is openly documented, exploring the fragment of compiled code for the most part boils down to reading the disassembled listing of programming code in the corresponding low-level language (which would be x86 assembly language for natively compiled executable modules and CIL for Microsoft .NET managed binaries, for example). In Microsoft Windows family of operating systems, Portable Executable (PE) is the universal file format for executable binaries (primarily EXE and DLL files).

While analyzing disassembly is far more complicated task than reading human-written source code of the same low-level language, it is still an affordable affair for the motivated researcher. In the process of reverse engineering compiler-produced executable modules, analystis presented to the number of natural obstacles (caused by the very fact that compiled binaries are targeted to efficient execution in corresponding software and/or hardware environments, rather than to being explored for logical re-analysis), but there are no artificial barriers specifically designed to hinder the process. Compiler authors are not motivated, obviously, to artificially impede the subsequent reverse engineering attempts. For this reason, such binaries are practically unprotected against reverse engineering endeavors.

But some developers have a need to make their compiled binaries (and sometimes the associated supplementary data files) as resistant to reverse engineering as possible. Mostly there are two categories of people that are interested in this: First are shareware software authors, who are concerned about software piracy – reverse code engineering is the primary discipline responsible for breaking legitimate licensing schemas which enables mislicensing, unauthorized reproduction and illegal distribution of software; The other group of people interested in protecting their code from being analyzed are malware authors – reverse engineering viral code modules is the key measure in fighting against malware. As a result, whole lot of anti-reverse-engineering techniques have been developed over time to prevent, or at least slow down, the analysis and/or modification attempts on executable modules. Dedicated software protection systems have been implemented, usually realized in the form of executable module packers.

The essence of executable packers. The executable packer is a software system used to compress, and in case of protectors – also protect (via payload encryption and various anti-reverse-engineering techniques), the compiled binary files. They originated to minimize executable file size in an effort to reduce associated disk space and network bandwidth usage. Packers can be divided into two major categories: the ones designed for executable compression only, and the ones that focus on executable protection from reverse code engineering, incorporating variety of appropriate techniques in addition to compressing the binary module. While no strict definitions exist, the latter are commonly called protectors, and former are referred to as compressors, or more often – just packers. To avoid ambiguity, packer will be used as the general term throughout this paper, while compression-only packers will be referred to as compressors.

The packer generates new executable from the supplied unprotected binary, which has little to no similarity to the original. It grabs the original PE module and compresses (and optionally encrypts) its sections into an archive that is added to the newly generated PE file as a data section. Some sections, like resources, might be opted to be left untouched, e.g. to make them statically accessible from the binary file to external applications. Then the decompression stub is placed in the code section, with a new entry point pointing to it. When run, the decompression stub unpacks the sections from the archive to their designated memory locations (as specified in the original PE header) and takes all the necessary actions (like resolving imports, handling module relocation, etc.) to ready the original code for execution, and then passes the control to the entry point of the original (now decompressed) application. This entry point of the original executable, where the control flow is firstly transferred from packer's code to the source application, is called Original Entry Point – OEP, and plays a very important role in the reverse engineering process of the packed executable. Packed application's import table is also different from the original, and usually smaller – only consisting of imports required to unpacking stub. Application's original imports are reconstructed (though often not in the original shape) and resolved after unpacking and before reaching the OEP.

One resulted effect of even simplest executable compression is the complication of the reverse engineering process: packed binary files no longer contain original executable sections in their natural form, so the static analysis, as well as direct module patching, becomes practically impossible and dynamic analysis is complicated significantly. While compressor's job is relatively straightforward and merely resembles the steps pointed out above, protectors take the process much further by implementing numerous anti-reverse-engineering techniques to hamper the analysis and modification attempts even more.To impede the comprehension of unpacking logic, the protector's unpacking code is usually self-decrypting and layered – each chunk is decrypted dynamically right before its execution, also erasing the old chunks of code. The original executable payload can also be encrypted multiple times and decryption/ decompression spanned to several stages. While encryption key is embedded into the packed payload, this theoretical attack vector is not chosen typically by reverse engineers

(because decrypting logic is so complicated). Most of the packer's code is heavily obfuscated and armed with numerous anti-debug tricks, to ward off attackers before reaching OEP. After decompression and decryption of original executable in memory, the packer applies relocation fix-ups if required and resolves imports, constructing the required Import Address Table (IAT) replacement. Finally, the execution jumps to the original entry point of the application (saved during the packing process).

Most of the stronger protectors ensure that the application still maintains the dependency to the packer's body after OEP, e.g. to make dumping of application's sections harder. Code section of the original binary is usually modified during the packing process, applying code obfuscation, virtualization, import redirection, and other anti-analysis and anti-modification techniques, to ensure protection after passing control to the original application. Also, many software protection system vendors offer SDKs which enable software developers to integrate protector code at the source level, enabling applications to utilize packer's (heavily protected) code for various security-related functionality: from debugger detection checks to generating hardware IDs for licensing routines, for example. Most protectors also incorporate full-featured software licensing solutions (usually accessible via SDKs), embedded right into the packer's code.

Protection techniques. Since the first executable packer was released in public, circa 1998, the protection technology has been developing constantly to meet the new challenges. Today's protection systems implement so many different defensive measures beyond just packing the binary modules, that the collective word "packer" might seem a little underestimating, but that is just a general term used throughout the reverse engineering community. In this section I'll try to review some of the commonly used approaches against reverse code engineering, used by modern software protection systems. While primarily focused on natively compiled executable modules (produced by languages like C\C++ and Delphi), many techniques will be relevant, or at least conceptually compatible, to other software platforms (like byte-code based frameworks, for instance) which store compiled code of any type in PE binaries.

Packing consequences. As outlined above, even simplest executable compression automatically provides multiple benefits from the protector's perspective:

Preventing static analysis – static disassemblers and decompilers are completely ineffective with packed binaries as the original sections are simply unavailable until runtime; Preventing direct file patching – even if the attacker manages to make necessary patching in memory, saving changes directly to the binary file is impossible, and even though "indirect" methods of patching (known as inline patching) make it possible, this significantly complicates the attacker's task. Concealing OEP – there is no standard way to determine when the execution reaches OEP after unpacking, and this is a significant barrier for reverse engineers; as to why – I'd like to elaborate on this subject a little bit further:

In almost every reverse engineering session, determining the original entry point of application and trapping the transition to it from the packer's code is the key requirement. There reason for this is twofold. Firstly, analyzing complex applications are frequently started from the entry point, otherwise comprehension of the whole picture could be hard or even impossible. In some cases, mostly with a GUI applications or DLL binaries, this might not be the necessity, but the ability to trace the execution from its very beginning is usually a big help for the analysis. Secondly, module dumping – which is the primary technique used for unprotecting packed applications after they are decrypted in memory, is possible at the OEP only, at least for the natively compiled applications. The reason for this is that after OEP the data section of file might be modified, which makes dumping dangerous (as the resulted section(s) will not be in original state anymore), plus the OEP should be known to be correctly specified in the PE header of the dump. As a side note: this usually is not the case with .NET assemblies protected with general-purpose (i.e. not .NET-specific) packers, where dumping is normally possible at any time after program's execution.

Debug prevention. Binary-level debugger is the main weapon against packers in hands of reverse engineer. Hence debugger detection and prevention at runtime is one of the central objectives of any protector. Variety of techniques exists to determine whether current process is being debugged or not. In case of debugger detection, actions vary from immediate process termination to concealed corruption of important runtime data structures to digressthe application from normal execution path, without reverser suspecting the reason for program's misbehavior. Many techniques are built upon the undocumented features of the platformarchitecture, so the implementations are often OS version specific.

Exploiting PEB fields. Process Environment Block (PEB) is the semi-documented runtime data structure available in every process, representing some of the process' important properties. Designed to be used by the application-mode code in the operating system libraries for internal needs, couple of its fields contain different values when the process is being debugged, and this can be exploited for debugger detection. While dedicated Win32 API exists for this purpose, namely kernel32.IsDebuggerPresent(), which determines debugger presence based on one of the fields of PEB, it is more common for the protectors to read values directly from PEB structure, because that API is well-known and easily patchable for most reverse engineers.

Inspecting process' heaps. When the application is being debugged, each heap of the process has certain flags set in its header, which subsequently result in appending special sequence of bytes to the end of each heap allocation– technique designed for easy buffer overflow detection. While the heap structure is officially undocumented, it has been researched by reverse engineers for various purposes. The protector can check for these flags or directly for the special signature of bytes, in any of the process' heaps to determine if the process was created in the debug mode. The main heap of the process (pointed to by the PEB. ProcessHeap field) is the common target of exploitation, but the other heaps can be inspected as well.

Utilizing undocumented low-level APIs. At the lowest level of Win32 API, in ntdll.dll module, there are couple of functions that, when called with appropriate (mostly undocumented) arguments, provide interesting information about debug session, if present. ZwQueryInformationProcess() and ZwQueryObject() are such API functions, that indirectly, though reliably, can be used to trap the presence of active debugger.

Checking for debug privileges.Debuggers need to acquire SeDebugPrivilegeright to debug another application. The ordinary applications usually do not have this privilege present in the access token of the process, but when the

process is created by the debugger it usually inherits this privilege, and this provides another possibility to determine debugger's presence. Protectors can check for this privilege, usually indirectly – attempting to execute actions that could only succeed with debug privileges. One common way is to try opening csrss.exe system process – an attempt that should fail from non-debugged applications. It is even possible to cause BSOD when SeDebugPrivilegeis available. Indirect ways of testing for the presence of debug privileges make this method a powerful debugger detection trick.

Manipulating SEH mechanism.Structured Exception Handling (SEH) is the native exception handling mechanism for Windows. Variety of anti-debug techniques employ on it, because of its special characteristics related to debugging. When the exception is generated, it is firstly passed to the debugger – if such is attached to the process. The debugger can then either consume it, or ignore – pass it to the application's event handler. The protector can install its own exception handler and then deliberately generate some exception, e.g. via dereferencing invalid pointer, dividing by zero, or even using kernel32.RaiseException() (or lower-level ntdll.RtlRaiseException()) system API. If the debugger will "swallow" (i.e. not pass to the application) the exception (and this is usually the default behavior), protection code will deduce debugger's presence.

In such cases, when the application expects to receive these exceptions, the debugger can be configured to pass all or specific exceptions to the program, but protectors have more tricks to circumvent this. The frequently used API kernel32.CloseHandle() (more precisely the lower-level ntdll.ZwClose() function it is based onto) will throw EXCEPTION_INVALID_HANDLE (0xC0000008) exception when passed an invalid handle value, but only when the debugger is attached. The protector can monitor for this behavior by deliberately passing the invalid handle to the function. In such case, in order to avoid detection, the attacker should not pass the exception to the program.

Another interesting method is generating DBG_PRINTEXCEPTION_C exception, which – as opposed to most of other exceptions – is unconditionally consumed by many debuggers, because it represents the means of passing the diagnostic textual messages to the debugger (functionality wrapped by kernel32.OutputDebugStringW() and similar APIs) and hence is not expected to be of any use for the application code itself.

One more interesting anti-debug trick is installing the last-resort exception handler via kernel32.SetUnhandledExceptionFilter() system API. While normally catching unhandled exceptions (purposely generated by the protector) when the application is not being debugged, when the debugger is attached to the process such handler will not be called, even when explicitly instructing the debugger to pass exception to the program.

Structured Exception Handling mechanism can also be used to manipulate the execution flow of the application, complicating the analysis process for the reverse engineer. Variations of discussed techniques can be combined to significantly complicate the debugging session for the attacker.

Breakpoint detection. Software breakpoints are usually realized in form of the one-byte INT3instruction (opcode 0xCC), which causes the processor interrupt that is handled by the debugger. When placing such breakpoint, first byte of the target instruction is overwritten with 0xCC byte, and when it gets hit the debugger restores original byte to continue execution. Memory CRC checks can be effectively used to detect software breakpoints, because code has to be modified when placing them. First bytes of known internal or imported functions, as well as small critically important code fragments, can be checked directly for the value of this opcode. Checking the return address of current function for this value is a very effective anti-tracing technique, as the debugger usually places such breakpoint after target function when performing the "step-over" functionality.

When software breakpoints are being detected, attacker might opt to use hardware breakpoints instead. They are implemented by means of special debug registers of CPU, where up to four memory addresses can be stored to be monitored not only for code execution, but for read or write memory access as well. Detecting hardware breakpoints is possible by inspecting the appropriate debug registers. This can be accomplished by retrieving the CONTEXT structure of current thread, either by explicitly calling the ntdll.ZwGetContextThread() API with CONTEXT_DEBUG_REGISTERSflag, or by artificially generating any exception and receiving pointer to the CONTEXT structure as indirect argument in the associated exception handler.

Alternatively, instead of detecting breakpoints, protector can try to blindly erase themperiodically, without even checking whether they exist. In case of software breakpoints, this is possible by overwriting important code fragments, desirably whole code section, from the file on disk. This will make all software breakpoints in that fragment disappear. With hardware breakpoints, the debug registers can be nullified to achieve the same results.

Execution timing. When the debugger intervenes in the program's execution, for example when processing breakpoints or generated exceptions, the execution is usually slowed down significantly. The protector can measure the amount of time spent for the execution of important code fragments, and deduce if the process was suspended meanwhile – presumably by the debugger. RDTSC assembly instruction can be used for such purposes, which returns internal CPU timestamp counter stored in one of its MSR registers. Alternatively, kernel32.GetTickCount() API can be used to get the system's uptime counter, or it can even be read directly from the specific user-mode memory location where it is constantly updated by the operating system's kernel code. When carefully concealed by code obfuscation techniques, execution time monitoring can become very effective debugger prevention measure.

Hiding the thread from debugger.The ntdll.ZwSetInformationThread function is officially documented as a routine that can be used to change the priority of a thread. But when it receives the undocumented HideThreadFromDebugger (0x11) info-class value as an argument, it sets the appropriate flag in the thread's ETHREAD kernel structure, causing all debug events associated to that thread to be discarded without passing them to the attached debugger. This means that the debugger will no longer be notified about breakpoints hit, or generated exceptions, for example. In case of the application's main thread, it won't even receive the process termination notification. This will effectively render the debugger useless for the most part.

Avoiding debugger attach. Sometimes, especially when the unpacking code of the protector is hard for the reverser to trace and finding OEP is not required,it might be preferred to attach the debugger to thealready running process, instead of starting the application under the debugger. When this happens, the operating system creates

a new thread in the target process, with an entry point at ntdll.DbgUiRemoteBreakin() routine which calls ntdll. DbgBreakPoint() function consisting of only one instruction – the INT3 software breakpoint. When this breakpoint is triggered, the first notification is sent to the debugger, the newly created thread terminates and the attach procedure is complete. In order to avoid attaching the debugger, the protector can patch either the DbgUiRemoteBreakin() or the DbgBreakPoint() function, and redirect to the kernel32. ExitProcess() API for example. In such scenario the application process will terminate as soon as any debugger will attempt attaching to it.

Self-debugging. The application can spawn a copy of itself as a new processand attach to it as a debugger. The real application logic will be executed in the child process. As there can't be more than one debugger attached to any process at the same time, the attacker will not be able to attach its own debugger to the child. Terminating the parent process will result in termination of the child process. This technique is not too hard to circumvent in its trivial implementation, by injecting the code in debugger process and detaching it from there, but some realty powerful protection mechanisms can be built onthis concept. One such example is the "Nanomites" technology from the Armadillo software protection system. When protecting, it replaces the conditional jumps in original application's code section with INT3 software breakpoints, and then uses debugging at runtime to control the child process. When such breakpoint is hit, the parent process that acts as a debugger determines the destination of the conditional jump and the exact type of branching instruction (from its heavily encrypted tables), and whether it should be taken or not (based on the value of the flags CPU register) and then, modifying the instruction pointer, resumes the execution flow accordingly. In this case it is not easy to detach the parent (debugger) process, as the child process simply cannot function without it.

Targeting specific debuggers.The number of popular debuggers most frequently utilized by the professional reverse engineers can be counted on the fingers of one hand. Nowadays just two of them are used in vast majority of the reverse engineering sessions. While most of the anti-debug techniques are more or less universal, in a sense that they can be equally successfully used for detection of any application debugger, some methods can be uniquely designed to target the specific one. This might be the bug exploited in the specific debugger, or the global operating system objects created by it, for example. One trivial (but by no means ineffective) method is enumerating and inspecting the window names in the system, in a search for a predefined text, characteristic to the specific debugger.

Dump prevention. To unprotect the packed applications, in many cases it is necessary to "unpack" them, i.e. to recover the application in its original form – removing the protector's cover, in order to ease the analysis process and/or make the permanent binary patching possible. The key to this job is the process dumping – grabbing the image of the process from the memory after its original sections have been fully decrypted there. Usually the dumping is only possible when the execution is paused at the OEP, for the reasons already explained previously. If the protectors did not retain any dependencies to the application after unpacking it (just as compressor packers work), it would be easy to unprotect them just by dumping the module's sections from the memory after reaching the OEP. But today's protectors employ variety of techniques to prevent or at least complicate this process. Anti-dumping mechanisms are built into every strong protection system.

Damaging the PE header.PE header is the crucial part of any PE module and describes the binary's contents and characteristics in detail. It is mostly utilized during the module loading phase, when the operating system's loader reads various parameters from it, and is rarely referred to afterwards. For successful dumping various fields of the header are read by the dumper programs, and they can be damaged by the protector to prevent dumping. Some of these fields are completely ignored by the OS loader (they serve the diagnostic purposes only), but not ignored by many debuggers and process dumpers, which means that such binary can be loaded and run successfully by the operating system, while unexpected errors might occur when trying to dump or even debug them. The even more radical approach would be erasing the whole header, especially the section table, after the module has been loaded, to neutralize many dumpers that rely on it, but this method isless frequently used by the protectors, as some APIs will still need to reference it at runtime, e.g. for loading resources from the module.

Damaging the SizeOfImage parameter of the module. The Ldr field of the PEB structure points to the PEB_LDR_DATA structure that contains the list of all modules present in the process and their parameters. Its InLoadOrderModuleList member contains the list of loaded modules, sorted by the time of their loading. Each of the modules is described by LDR_MODULE structure, SizeOfImage field of which represents the size of the module in memory. It is set by the OS loader from one of the fields of the PE header when the module is loaded. Many dumpers rely on this value to determine the region of memory that belongs to the specific module, and damaging it by specifying some enormously large bogus number would most probably render them unusable.

On-demand code decryption. Most of the packers fully decrypt the whole application before passing control to the OEP, and this is the major weakness of the whole concept of an executable packing in general. But there exist some implementations where this is not really a case. It is possible to dynamically decrypt only those parts of the program that need to be executed or accessed next, and then preferably erase them later – without exposing the whole picture of the application's sectionsto the attacker at any given moment of time. Each section of the executable binary is usually one atomic structure, not quite divisible by any standard means, so there aren't many ways of implementing this concept. Generic method that can be used to protect any binary with this technique is based on the memory guard page mechanism of Windows. A guard page provides a one-shot alarm for memory page access – memory region can be marked as guard pages, and any attempt to access them will cause the system to raise a STATUS_GUARD_PAGE_VIOLATION (0x80000001) exception (turning off the guard page status at the same time). The protector can intercept this exception, check the destination address, and if it falls in the specific range, decrypt the appropriate part of the protected section. To make dumping and analysis even harder, it is possible to erase old pages, e.g. when new pages will be triggered, but the performance penalty will thenbe an issue in many cases.

Code stealing.Each PE module consists of the PE header and multiple sections. When dumping the module

from memory, usually only the sections which make up the module are grabbed, not any other memory allocations of the process. The protector can steal the chunks of the code from the code section, and place them in the memory buffer (with page execute access) allocated dynamically. There will be jumps placed to and from those stolen code buffers connecting them to the code section, so the application's execution will not be affected by such modifications. The place of stolen bytes in the code section will be filled with garbage, and the relocated code will be usually obfuscated and interleaved with garbage instructions, to complicate the restoration of the code in its original location. As a result, when the attacker will dump the module, the code section will contain "holes" in place of stolen code chunks, consisting of jumps to the nonexistent memory allocations. The stolen code can be not only obfuscated, but sometimes virtualized or interleaved with anti-debug checks, creating the dependencies to the protector's code section. This techniqueis employed by many advanced protections systems and makes dumping extremely difficult for reverse engineers.

Manipulating imports mechanism. Practically all PE modules import number of functions from some dynamic link libraries (DLLs). This is accomplished by the well-defined imports mechanism. When loading the executable binary, Windows's image loader parses the import table and resolves imports by loading referenced libraries and constructing Import Address Table (IAT) – an array of pointers containing the virtual addresses of the imported functions. All invocations of the imported functions in the code are made through these pointers. In order to complicate the damping process, the protector can eliminate the import table and save the names of referenced DLLs and corresponding imported functions in its own, obfuscated format. During unpacking, before transferring control to the OEP, it will determine and resolve all the imports, constructing the IAT in a similar manner like the operating system's image loader does. By strippingthe original import table, the module's dump will not be possible to be loaded by the OS's image loader, as it will not have the information necessary for building the IAT. The attacker can recover this information by scanning the IAT after it has been built by the protector and determining the names (or ordinals) of the functions (and their exporting modules) pointed to by the entries of the IAT. In order to avoid this, some protectors employ the import redirection – in the allocated buffers they place obfuscated trampoline routines, which do nothing but call one of the imported functions. Then they place the addresses of these redirection routines in the IAT, instead of original function addresses. In this manner, original function names can't be automatically determined by just scanning the IAT – the attacker will have to analyze each and every such entry of the IAT to deduce to which function of which module they are actually redirected. To complicate the import reconstruction even more, some protectors completely eliminate the use of IAT and instead patch the code section directly replacing the calls to the imported functions by the calls to their own one particular functionwhich serves as the universal redirector.The redirector routine is same for all imported functions – it determines the correct original function address based on the return address placed on the stack. It is always heavily obfuscated and might contain occasional anti-debug checks. Recovering the redirected imports is the challenging job and very time-consuming even for the skilled reversers.

Patch prevention. Patching refers to the process of modifying the compiled binary modules, in order to alter the behavior of its execution. Modules can be patched statically by permanently modifying them on disk, or dynamically – by modifying them in memory at runtime. Unprotected executable modules, in a form they were produced by the compiler (more precisely by the linker), are easily patchable: desired modifications can be usually made by directly patching appropriate bytes in code section, while addition of whole new executable sections to the PE binary is also possible for some complex patching scenarios.

However, there exist several methods to complicate the patching process. Even simple executable compression, used by almost every packer, including those not specifically designed for anti-reverse-engineering protection, makes direct static (on-disk) patching impossible. Original PE sections are no longer available in packed executables, so there is nothing to patch in a file on disk. Attacker can still patch such modules dynamically in-memory (either by another application, or by embedding the dynamic patching logic into the file itself - technique known as inline patching), or by unpacking the packed modules into the original form to enable direct patching. None of these are quite trivial to accomplish, though.

While not the only defense against patching, primary anti-patching measure employed by protectors is the verification of control sums (checksums) of important fragments of the binary module at runtime. This technique, also referred to as CRC checks, allows detection of even a single-bit modification of the binary (or some of its regions), and can be applied to detect both on-disk and in-memory patching attempts. In its simplest implementation, CRC check implies calculation of some kind of checksum (by any cryptographic hash algorithm) of the compiled binary (or some of its fragments, like code section, for example), saving of this checksum along with (usually inside) the binary, and verification of it at the desired moment at runtime (e.g. during module initialization). Digitally signing the module, and verifying this signature at runtime, is one realization of this concept. In more complex implementations, precalculated checksums can be used as a key in a symmetric encryption process of some kind. For example, protectors could encrypt the code representing the next layer of unpacking routine with a checksum of the previous layer. That way, any modification of previous layer (and for example, even placing the software breakpoint inside such code section during debugging session results in a modification of that section in memory, as outlined earlier in this paper) would make decryption of the next layer impossible.

Protectors incorporate CRC checks to thwart the inline patching, dynamic patching, and unpacking attempts. Attacking the CRC checks usually involves either falsification of stored checksum by overwriting it with desired value, or patching the checksum verification routine itself. Concealing and complicating the CRC verification process is a major concern of protector authors.

Even stronger patch prevention mechanism, although significantly complex in its implementation and with some notable limitations, is the code virtualization technique, which will be discussed shortly.

Code obfuscation. In order to analyze compiled binary, reverse engineer has to read through the thousands of lines of the disassembly presented in a low-level assembly language, often heavily optimized by the compiler for the target execution environment. While

quite challenging task as it is, comprehension of execution logic of the compiled code can be further complicated by obfuscation techniques. Any program code can be theoretically rewritten practically unlimited number of times into different physical representations, yet retaining the original execution logic. Code obfuscation implies rewriting original code fragments into new representation with different instructions, replacing simple constructs with more complex, but logically equivalent code, retaining the original code meaning. For example, simple arithmetic operation, involving only a couple of elementary assembly instructions, can be rewritten into tens or even hundreds of elementary instructions, performing complex calculations, yet producing the same result.

Along with complicating the execution logic, number of neutral code constructs can be inserted randomly, which alter the execution state in numerous ways, but then revert the changes back to neutralize the effect, thus not changing the logical flow of execution at all. In addition to such "garbage instructions", some amount of random "garbage bytes" can also be inserted throughout the code in the places unreachable by normal execution flow. Such bytes will always be skipped by unconditional branching, or conditional branching instructions which will be executed when branching condition is always true – always skipping these bytes. If executed, such bytes would certainly make up the invalid instruction, resulting in a critical execution fault, but they will never be executed by the processor at runtime. On the other hand, disassemblers – especially those operating with linear weep method – interpret such bytes as parts of legitimate instructions. As a result, instruction parsing proceeds in a totally wrong direction, thus producing completely different assembly listing, compared to what will actually be executed.

Slightly decreased execution performance of obfuscated code, caused by increased complexity of execution, is often neglected, considering benefits the code obfuscation offers in a battle against reverse engineering.

While making the reverse engineering process much more time-consuming, deobfuscation – performing the reverse transformation of the code – is always possible to some degree. Many obfuscation techniques can be analyzed to the extent that even makes the creation of automated deobfuscation tools a real possibility. Nevertheless, code obfuscation remains to be one of the core concepts in reverse engineering field.

Code virtualization. One of the most powerful anti-reverse-engineering techniques employed by modern software protection systems is code virtualization: during binary packing the protectors translate chunks of original code into custom meta-language, which is then interpreted at runtime by appropriate virtual machine – pure software execution environment, which is part of the protector and thus – embedded into the binary itself. The concept is similar to that of bytecode-based programming frameworks like Java and .NET. But unlike to them, instruction set is not documented publicly, and virtual machine is heavily obfuscated and armed with numerous anti-analysis tricks, to complicate the reverse translation process. Not only does this make analysis (not to mention patching) of such code fragments orders of magnitude harder, but also chains the whole binary to its virtual machine (and hence – to the protector) as the module can't function without it, thus making unpacking impossible.

In more advanced protection systems, opcodes for every virtual instruction of the same virtual machine family are usually generated specifically for each particular binary that is being protected. As a result, opcodes are different in every protected binary and even if some opcodes were deciphered based on analysis of one such module, it would not be possible to identify same instructions by the same opcodes in another protected binary. One-to-one relation between virtual and real instructions is not a necessity: one virtual opcode might represent multiple assembly instructions, or, more frequently – one assembly instruction might be translated into several, more primitive virtual instructions.

Decompilation of the virtualized code requires in-depth analysis of the corresponding virtual machine, and this is one of the most challenging tasks in modern reverse engineering world. Once the virtual machine has been studied well enough, the mapping between virtual opcodes and the original instructions can be established, and the decompilation of such code would then be possible.

Due to the performance hit of the virtual machine and the code interpretation process, authors usually choose to limit the virtualization to the critically important fragments of the code.

David Nachkebia

ANALYSIS OF WINDOWS PORTABLE EXECUTABLE SOFTWARE PROTECTION SYSTEMS

SUMMARY

Reverse Code Engineering (RCE) is a process of analyzing software system, usually in a compiled state – without corresponding source code, in order to decipher its modus operandi and optionally – to even modify its behavior. While multiple use cases exist, more often than not RCE is utilized against the will of the authors of target software product. It is the primary discipline responsible for the analysis and prevention of malicious software on the one hand, and for tampering of proprietary software licensing schemas – on the other.Hence these two, quite unrelated group of software engineers – malware authors and proprietary software developers are the main victims of reverse engineering. As a result, a number of sophisticated protection techniques have been developed to prevent, or at least slow down the process of reverse engineering and unauthorized tampering of compiled binaries, and these anti-RCE countermeasures bring the challenge of reverse code engineering to the next level.

The article presents the analysis of anti-RCE software protection systems on Windows platform. It discusses their implementation details and various techniques employed for protecting compiled executable binaries from reverse code engineering. The essence of executable packers and their principles of operation are analyzed; various anti-debug, anti-dump, patch prevention, code obfuscation, and other complex anti-reverse-engineering techniques are uncovered.