Automated unpacking of executables using Dynamic Binary Instrumentation

Tadas Vilkeliskis
Department of Computer Science
Stevens Institute of Technology
tvilkeli@stevens.edu

August 13, 2009

Abstract

To create better protection solutions against malicious software (malware) first such software must be analyzed. Today application packing is very popular among the authors of malware. This paper briefly covers what protections are out there in use and what proactive solutions have been created. Also the paper gives a short classification of both packers and unpackers. Additionally an example of each class is given. Also an unpacking method based on MmmBop Dynamic Binary Instrumentation (DBI) engine is presented. The unpacking method was designed to automatically penetrate multiple packing layers and handle multi-threaded applications. The unpacker concentrates only on automatically finding original entry point and reconstruction of import tables or PE header is not covered.

1 Introduction

Computer software market is a very competitive place. Software writers and companies are constantly trying to protect their products from reverse engineering. In some cases the underlying code is the reason why a company is receiving revenue. However, today protecting program's code does not necessarily mean the protection of the intellectual property. Writers of the malicious software regularly use code protection techniques to evade virus scanners and hide intentions of the malicious program.

The code protection methods can be split into two categories: anti-static and anti-dynamic analysis protections. Anti-static methods includes a variety of code obfuscation techniques, such as code reordering, insertions of junk code, program flow modifications, encryption and many others [25]. Code obfuscation techniques make static analysis nearly impossible in most cases, because the tools we have today cannot handle all obfuscation tricks, especially the ones that use encryption or compression algorithms. That is where the dynamic analysis comes in. However, the dynamic analysis is not always easy to perform either. A lot of anti-dynamic analysis tricks have been introduced [31]. The most common one is the debugger detection and evasion due to the way protected code is analyzed. Usually a protected program use both anti-static and anti-dynamic methods simultaneously. This makes the code analysis even harder and very time demanding for a human reverse engineer.

This paper concentrates on packed executables and presents an alternative automatic unpacking solution. Additionally some packing techniques are shown and discussed from the perspectives of both resilience to reverse engineering attacks and cryptographic methods.

2 Basic packing and unpacking

This section will explain what a packer is and basic steps that are taken to protect executable code. We will also take a look what steps are necessary to obtain original code by unpacking a packed application.

2.1 Packing

Packing is a process of encrypting or compressing executable code and embedding that code into a newly created executable file. The new executable basically consists of two sections: the packed executable and the unpacking code. When we run packed executable the unpacking code runs first and it decrypts or decompresses original executable to the memory. Finally the control is transferred to the real program. This is a base for every packer, however some of them use a bit more complicated ways to unpack the original code.

2.2 Unpacking

Unpacking is a process of recovering the original executable code from a packed executable. The unpacking process usually consists of three stages:

1. Finding original entry point (OEP) and writing process memory to a file.
2. Creating or modifying Portable Executable (PE) [26] header for a file we have written in a previous step.
3. Reconstructing Import Address Table (IAT).

If we think about basic packing, then the original code is clearly visible in memory after the unpacking algorithm has finished its execution. Therefore it is possible to write the memory of the process to a file. The written file must be fixed because the code holds invalid references to imported functions. If we are dealing with more sophisticated packing methods, sometimes the whole executable image might not be visible in memory at the time and extra steps must be taken.

3 Packers and code protectors

There are so many different protectors out there. The protectors can be classified by their resilience to reverse engineering attacks into three basic classes: simple, intermediate and advanced. Simple protectors don't use any anti-debugging or anti-analysis tricks. They simply encrypt or compress original code and make it available at runtime. Intermediate protectors include various anti-debugging and anti-analysis tricks to make removal of the protector more complicated. Advanced protectors include much more sophisticated anti-analysis and anti-debugging techniques such as multiple threads or virtual machines.

Packers that we have today usually encrypt original executable instead of compressing it. Therefore packers can also be classified by the type of encryption algorithm and whether the incremental decryption is used or the whole executable is decrypted at once. Most of the packers include multiple encrypted layers. These encrypted layers are usually found at the beginning of the packed application and are used as a stepping stones toward real decryption algorithm. The stepping stones decrypt the code or the data that will be accessed later by the packer. The decryption algorithms used in the first steps in most cases are fairly simple. The decryption code can be only few bytes long (when obfuscation is removed) and use stream based ciphers. The decryption key can be embedded into the executable or calculated at runtime. Runtime keys in most cases are obtained from the buffer size that needs to be decrypted. This method is commonly seen in polymorphic viruses [53]. The decryption algorithm is small and easy to implement and it can easily evade signature-based virus scanners. In more extreme cases, the key can be obtained over the Internet. The actual decryption algorithm that is used to decrypt the original program can vary among simple stream ciphers and more sophisticated strong cryptographic solutions.

If we look at the example of Yoda's Protector [17] we will encounter multiple encrypted layers that are decrypted with different algorithms. First two decryption loops are shown in Table 1. The first decryption loop hides packer from the static analysis tools, such as a disassembler. The algorithm is a stream cipher where the key is a lower byte of the size of the data (for a step-by-step decryption refer to Appendix B). The key can have only 256 different combinations and its value is changed in a linear fashion. If more than 256 bytes must be decrypted the key values will repeat. Further more the key is combined with additional constant data that is utilized in bitwise operations. Other similar loops decrypt data, such as strings that are later used to get addresses of a particular Windows Application Programming Interface (API) functions. More interestingly the real program is encrypted with a strong cryptographic algorithm. Specifically, RC4 [18] was used in this case. The key for the RC4 cipher was obtained from a MD5 [45] hash function. The MD5 hash function takes 32 bytes embedded in the executable and creates a hash value. This value is used later on in RC4. Yoda's Protector does not implement these cryptographic algorithms by itself but instead it takes ones provided by the Windows Cryptography API [14]. Other packers might use different cryptographic algorithms such as AES, Blowfish, Twofish, Serpent, RC6, Mars [19, 47, 48, 20, 44, 23]. Also a custom random number generator can be implemented with Linear Feedback Shift Register (LFSR) [37]. SFX Creator [41] is an example of such packer.

The following are few packer examples classified by the resilience to reverse engineering attacks.

3.1 UPX (simple)

UPX [12] is a simple packer whose purpose is to compress executable to make it smaller. UPX uses a UCL data compression library [40]. The UCL has a high compression ratio and also, unlike LZO [39] or similar compression algorithms the UCL does not require additional buffer for decompression. Thus, the decompression can be performed on-the-fly with some performance penalty. The newly created executable consists of UPX*[1] and resource sections. Usually UPX section name with the highest number is the section that holds compressed

[1] * denotes a number 0, 1, 2, ...

executable and decompression stub. Other sections are used to store decompressed executable image. UPX can be very easily identified due to the decompression stub—it always remains the same (only few addresses are changed). This packer also does not include any anti-debugging or anti-reverse engineering tricks since it was not designed to protect executables from reverse engineering.

3.2 Conficker worm (intermediate)

Conficker [42] is not a protector, however it includes few interesting protection mechanisms. Conficker comes as a dual-layer packed executable. First layer is removed with a custom decryption algorithm. Also the first layer has a lot of polymorphic code and some anti-debugging and anti-tracing techniques. Early versions of Conficker used `IsDebuggerPresent` API call to check for debugger and `RDTSC` instruction to see if code is being traced. In other versions Conficker has changed debugger detection with `CloseHandle` API call. `CloseHandle` has an interesting property—when the process is being debugged and invalid handle is passed to the function, `CloseHandle` will raise an exception. However, if the process is not being debugged the function will return only error code and no exception will be raised. The virtual memory allocation function, `VirtualAlloc`, is called to allocate memory page for a second protected layer. The newly allocated memory page is made executable after the whole second layer is copied into it. Execution continues at the second layer. The original code from the second layer is decompressed with UPX. After the decompression, the original executable image does not have a PE header, therefore a PE header template must be used in order to perform static or dynamic analysis later.

3.3 Armadillo (advanced)

Armadillo is a protector that incorporates both license manager and a program packer [43]. Armadillo uses a sophisticated packer. The packer includes few decryption stages. The initial stage decrypts code that is used to detect a debugger. Also Armadillo makes static analysis very difficult due to advanced code splicing and jumps into the middle of other instructions. This packer also performs runtime self-patching, where some bytes are replaced with `NOP` instruction. Therefore static normalization tools such as [24] will fail because code cannot be rearranged or junk instructions removed. Armadillo also makes use of advanced anti-debugging trick which is illustrated in Figure 1. To defend against debugger Armadillo creates an extra thread that attaches to the running process as a debugger. Windows OS allows only one debugger to be attached per process. Thus debugging would fail because the debugger thread could not be created or a debugger could not be attached. Armadillo replaces some jump instructions with `INT 3`, a software breakpoint also known as nanomite in reverse engineering community. When a software breakpoint is encountered the operating system will raise an exception that is handled by the debugger thread. The debugger thread searches for the exception address in the exception address lookup table, and takes the address where the child thread should continue its execution. Armadillo also adopts decryption on demand. Virtual pages are protected with `GUARD_PAGE`. The guard page exception will be raised when a protected page will be accessed and handled by the packer leading to the unpacking of the page. More in depth analysis of Armadillo 4.20 can be found in cited work [21].

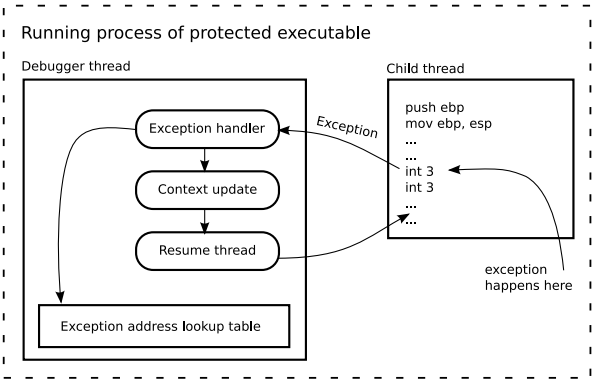

Figure 1: Armadillo's self-debugging and control transfer using nanomites.

3.4 Other protectors

Some protectors use virtual machine code to obfuscate the original code. VMProtect [13] replaces the original code with a random instruction set which is executed through virtual machine. Unlike VMProtect, Themida [11] use virtual machine to protect decryption algorithm. Themida also makes reconstruction of original program much more difficult by obfuscating imported functions. Imported functions are copied to a new memory page where they get disassembled, obfuscated and reassembled. Other implementations such as Defender [30, pp. 370-419] decrypt code only before its execution and reencrypt it afterwards. Therefore, writing the memory of such application to a file might be a difficult task. Some packing solutions include ASPack and ASProtect [1], Exe Protector [3], PELock [9], MPRESS [7] and others.

Though packers are extremely popular among the family of Windows operating systems, few executable pack-

ing solutions were created for GNU/Linux systems too. Shiva [10] was created to protect Executable and Linkable Format (ELF) [4] executables that was later cracked by Chris Eagle [28]. Other protectors for GNU/Linux include Burneye and Burneye2 (objobf) [2].

4 Related work

There have been few generic unpacking approaches released to the public. This section will provide a classification of unpackers and additionally look at example of each class.

4.1 Signature-based unpacking

Signature-based unpackers are the most simple ones. Usually entry point signatures are created for every packer available. The entry point of a packed application is scanned and if a signature match is found the correct unpacking algorithm can be applied. PEiD [8] is one of the most popular tool for identifying packing algorithm. Even so, there is one big disadvantage of signature matching—database must be constantly updated. Also it will not work with polymorphic packing engines, where the code looks different every time. For such code new signature must be created with every polymorphic iteration of the same application.

4.2 Heuristic-based unpacking

Unlike signature-based unpacking, heuristic detection does not identify what packer was used, but extracts features of packers and can perform unpacking generically.

Universal PE Unpacker (UPEU) is a plug-in for IDA Pro [6] that use behavioral heuristics to identify OEP. UPEU places a breakpoint on `kernel32!GetProcAddress` assuming that this function will be called before OEP, when IAT is created. However such methods might not always work and can give many false positives. Programs like Defender or Themida packer make a copy of the code of imported functions and place it to a newly allocated memory page. Therefore the breakpoint will never be hit.

OmniUnpack is a generic unpacking tool implemented as a kernel driver [36]. OmniUnpack monitors execution of the program and records memory locations that were written. If the memory location that was previously written is executed, OmniUnpack invokes a malware scanner to analyze the memory page. Finally it terminates the execution if the memory page is identified to be malicious. OmniUnpack monitors memory access by forcing a write-xor-execute policy—the memory cannot be writable and executable at the same time. However the write-xor-execute policy implemented in OmniUnpack can be easily evaded by using dual mappings [50]. A dual mapping is created when we allocate two virtual pages that point to the same physical location, where one virtual page is writable while another is executable. OmniUnpack is a good addition to malware scanners, yet it was not designed to extract packed code.

4.3 Trace-based unpacking

In trace-based unpacking a program tracer is implemented so it could follow the execution flow of a packed program and perform code analysis when needed.

PolyUnpack was created as a tool to extract hidden code [46]. It is implemented as a debugger which comes in two versions: standalone and a plug-in for OllyDbg debugger [56]. PolyUnpack is based on behavioral analysis approach and expects a particular action to place (e.g. execute modified memory). It creates a static view of the program and performs dynamic analysis. During the dynamic analysis program is stopped after every instruction and every memory write is logged. The execution is halted after the instruction, that is going to be executed, is not in the static view. The differences between the static view and dynamic view can be then extracted. However, PolyUnpack has few drawbacks. Since it is implemented as a debugger it is prone to detection. Also PolyUnpack does not enter the Dynamic Link Library (DLL) calls assuming that the code in a DLL is legitimate. For instance, Waledac worm overwrites few bytes in `ntdll.dll` with a call to its own code. Therefore PolyUnpack would be successfully evaded by this technique. In addition, PolyUnpack cannot recognize whether the halting instruction is the original entry point of the program.

Renovo [34] monitors instruction pointer jumps to the memory region that has been modified in previous execution steps. Renovo uses a clean and dirty flags (i.e. shadow paging) to mark the modified memory. When program performs a memory write instruction the destination address is marked as dirty. When instruction pointer jumps to one of these dirty regions Renovo determines a packed layer and instruction pointer points to OEP. To handle multiple packing layers Renovo initializes memory as clean again and repeats the process until the time-out is reached. However, by using timeouts Renovo makes itself vulnerable to time-out attacks where malicious software can intentionally give up the CPU and successfully evade Renovo.

4.4 Other tools

OllyBonE [51] is a combination of a kernel driver and a plug-in for OllyDbg. The kernel driver allows break-on-execute at page level. The user must select which page he wants to protect. When the break occurs the debugger gets notified. Such technique can be used to identify OEP because usually original code is extracted to different code section and memory page.

Rotalumé [49] specifically targets packers that use code emulation to hide original code. *Rotalumé* executes the emulated application and records the entire x86 instruction trace generated. Then it applies dynamic data flow and taint analysis techniques to these traces, data regions containing the bytecode program and extracts information about bytecode instruction set. By doing that *Rotalumé* is able to identify fundamental characteristic of decode-dispatch emulation and generate both an instruction trace and control flow graph.

Ether [27] is a program tracer based on application of hardware virtualization extensions and it stays completely outside of the target OS environment. Ether use Xen Hypervisor [16] as a base for virtualization. The tracer has two components. Xen Hypervisor component detects events in the analysis target such as system calls, instruction execution, memory writes and context switches. User-mode component acts as a controller for the hypervisor's component. Execution is monitor via trap flag, to find modified memory Ether employs shadow memory tables and system calls are monitored with hooks. Even Ether achieves higher transparency than any other tool that is available today it still is prone to timing attacks also it requires specific hardware that support hardware virtualization. Finally, due to its fine execution trace it not suitable for real-time applications.

5 Our unpacking approach

5.1 Design

The unpacking method we present here is based on MmmBop DBI engine [22] and is targeted at Windows platforms. The unpacking method initially concentrates only on finding the OEP. An automatic reconstruction of import tables or PE header is not covered here, therefore it must be performed manually. The extended version of MmmBop is shown in Figure 2. Like MmmBop, our unpacker will consist of two main modules: injector and unpacker.

5.1.1 Injector module

The injector module is responsible for injecting the unpacking module into the packed application's address space. This can be achieved with `kernel32!CreateRemoteThread` and `kernel32!WriteProcessMemory` API functions after creating a suspended process [35]. If injection was successful, the injector will invoke unpacker's initialization routine which will prepare unpacker for execution and install certain API hooks (a proxy function between the caller and the callee). Finally, the injector will resume the suspended process and terminate itself.

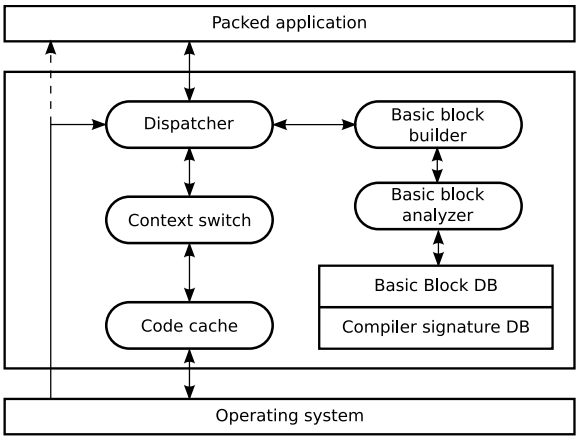

Figure 2: Extended MmmBop DBI engine.

5.1.2 Unpacking module

The unpacking module consists of 6 components: the dispatcher, context switch, code cache, basic block builder, basic block analyzer, and basic block and compiler signature databases. The context switch, code cache and basic block builder will be reimplemented to have the same functionality as the corresponding components in MmmBop DBI engine. That is to say, context switch will switch from currently running thread's context to an internal unpacker's context to create stack and CPU transparency. Basic block builder is responsible for creating a basic block by disassembling code from the starting address of a basic block till next control flow changing instruction. And code cache natively executes basic block instead of emulating it.

5.1.3 Internal databases

There will be two databases available in the unpacker's module: basic block and compiler signature databases.

The basic block database is constructed at runtime during instrumentation. This database will hold $(address,\ checksum)$ pairs, where address is the starting address of a basic block and checksum—the code checksum of the basic block. The $checksum$ field will hold the last checksum value that was calculated before

performing basic block analysis.

The compiler signature database will be used to identify OEP of a packed program. It will hold entry point signatures of many various compilers. Some of the signatures can be obtained from PEiD and others should be created manually to match as many compilers as possible.

5.1.4 Basic block analyzer

There is no easy way to identify address of OEP. Currently most of the tools rely on assumption that unpacked code will be extracted to a particular memory range or after the control is transfered to a modified memory. The basic block analyzer is responsible for identifying OEP. Initially the identification will be based on the compilers signature matches. If we have a signature match the process memory will be written to a disk. Additionally, basic block analyzer could perform statistical analysis such as sliding window randomness test [29] of the process memory, to see if there are still some packed code left. The basic block analyzer will be invoked by the basic block builder only when there is a basic block checksum mismatch or the address of the basic block is not in the basic block database. This is useful if we are in a loop and the loop does not contain self-modifying code, thus we will save few CPU cycles on signature matching. Also compiler signatures serve another purpose. They will help to penetrate through multiple packing layers.

It is also possible to implement an alternative OEP identification method. The basic block builder emulates code of a basic block to figure out whether the basic block performs self-modification. Therefore, Hump-and-Dump could be used to find possible addresses that are close to original entry point [52]. Hump-and-Dump is a very primitive algorithm—it counts how many times an instruction at the particular address was executed. Usually, when we deal with a packed or an encrypted executables some addresses will be executed more often (we get a hump) than the others before transferring control to the original program. This is a cause of a decryption or decompression loop. One inconvenience of using the Hump-and-Dump is that you have to provide an instruction count threshold, meaning that we only have a hump when we reach the threshold. After reaching the threshold the basic block analyzer can be invoked on a particular block to see if there is a known signature match. The Hump-and-Dump can be used to increase unpacker's performance because signature matching is not performed very often. However, there might be cases when a threshold will never be reached if it is too high.

5.1.5 Dispatcher

The dispatcher is the most important component in the unpacker's module. It is responsible for retaining control of the program we are instrumenting and making decisions about what to do next.

To retain control the unpacker's initialization routine will install `ntdll!NtContinue` [38] and `ntdll!KiUserExceptionDispatcher` [32] hooks. Hooks for these functions were also implemented in MmmBop to defend against tricky control transfers through exceptions or direct thread context changes. However, to hook these two functions is not enough to handle multi-threaded applications. Therefore, I propose a possible solution for handling multi-threaded applications.

One of the important things when instrumenting a multi-threaded application is to intercept a context switch between threads. However, the context switch is performed by the Windows kernel and there are no kernel-mode to user-mode notification routines (`ntdll!KiUserApcDispatcher` [33] is called if we are synchronizing objects through `WaitForSingleObject` or similar functions. However, it is not called if we are dealing with asynchronous operations) that could be intercepted. This is a huge obstacle when whole unpacker is implemented in user-mode. A possible solution is to place a hook at address where thread starts executing. In Windows OS every thread start their execution at `kernel32!BaseThreadStartThunk` which is a system-wide location. However this location might vary among different versions of Windows. It is located at `kernel32!CreateThread+X` where X is an offset. On my testing environment, Windows XP Professional SP 3, the location of `kernel32!BaseThreadStartThunk` is at `kernel32!CreateThread+0x22`, `7C8106F9`. When BaseThreadStartThunk starts executing the `EAX` register holds the address of a thread function supplied by a user. This function is the actual code that user wants to execute. Thus, by installing a hook at BaseThreadStartThunk we can insert control retention code at the beginning of the function.

There also might be cases when one thread wants to change another thread's context. To handle such situation `ntdll!NtResumeThread` and `ntdll!NtAlertResumeThread` [38] must be hooked. The thread must be suspended so its context could be changed and resumed afterwards. The ResumeThread hook will check the context structure before resuming a thread and insert control retention code into the location where the thread will continue its execution.

Finally, since the dispatcher is the unpacker's entry

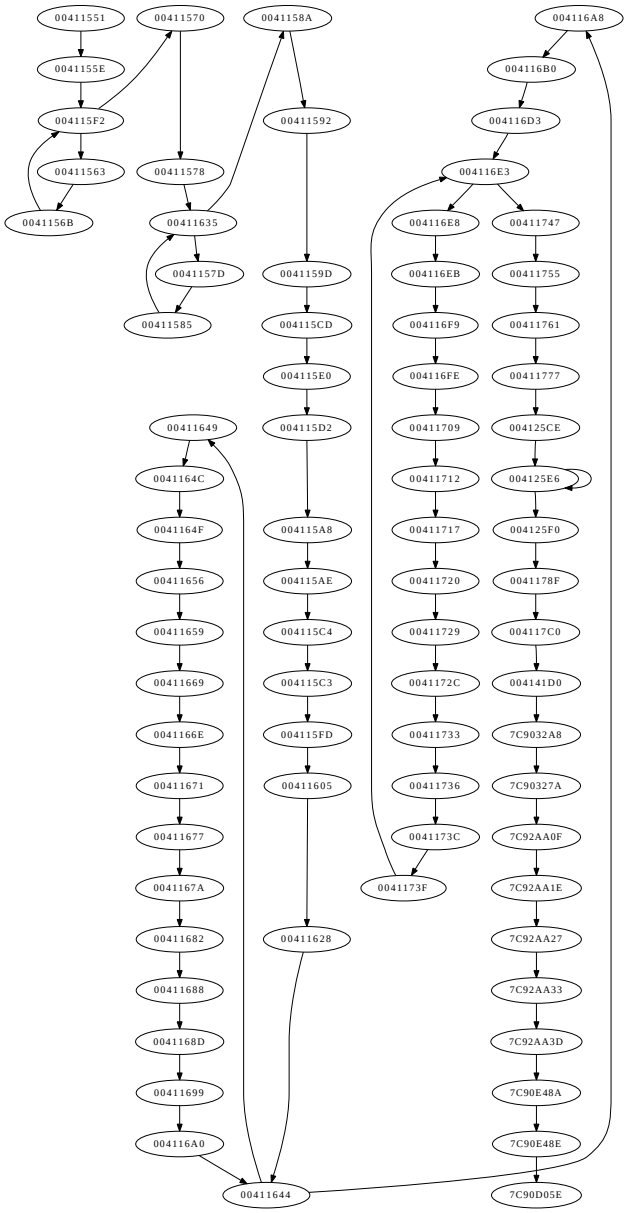

Figure 3: Control flow graph. Each node represents a starting address of a basic block.

point and it can be accessed by multiple threads at the same time the *Same Instrumentation Code Multiple Data* design will be used [55] to implement unpacker's module. Every thread will share same instrumentation code and thread related data will be accessed through thread table. To achieve this the instrumentation code will be surrounded with multi-threading preamble that will select appropriate data structures using thread's ID.

5.2 Implementation

The tools used in the research for analyzing packer formats and malicious software were the following: IDA Pro disassembler/debugger, OllyDbg and Debugging Tools for Windows (WinDbg) [15]. Most of the code was analyzed at runtime with a debugger. The very same tools were used in the unpacker's development to debug the code. The unpacker is written in C++ with Microsoft Visual C++ compiler. It was implemented as a DLL to make the injection process more simple.

The unpacker is in early prototype stage. Nevertheless, the ongoing implementation of the unpacker was constantly tested with a simple executable that was packed with Yoda's Protector. Yoda's Protector provides a great testing environment due to multiple encrypted layers, control flow changes via exceptions and use of `NtContinue`. The unpacker still requires many features to be implemented. Currently the unpacker has achieved full stack transparency by allocating new memory, modifying `ESP` register and Thread Information Block [54] structure in the process's memory so that no modifications could be caused to the stack of the real program. Additionally a simple control flow tracing can be performed. The following control changing methods are supported:

- JMP, Jcc (conditional jumps), RET and CALL instructions.
- Structured exceptions.
- Context changes via `NtContinue`.

The histogram in Figure 4 shows the data that was extracted with the unpacker. Three spiked intervals from the figure represent three loops. First two loops are used in decryption and span across multiple basic blocks due to the code transformations made by jump instructions. The third loop performs some sort of checksum calculation and takes only one basic block to do that. Table 2 presents the first few bytes that get decrypted by the first decryption loop, `00411644–004116A3`, that is shown in Table 1. The decrypted code also utilize additional jump-in-the-middle-of-instruction obfuscation technique. For example, the instruction at address `004116a8` makes a call to `004116b0` where `33` is an opcode for `XOR` instruction. This is a very common technique used by many packers to fool disassemblers and decrease code readability.

The unpacker was also able to extract data for generating a control flow graph, Figure 3. The graph shows all the basic blocks that have been executed up to `NtContinue` function. Additionally, one can clearly see all the decryption loops and passing the jump-in-the-middle-of-instruction obfuscation.

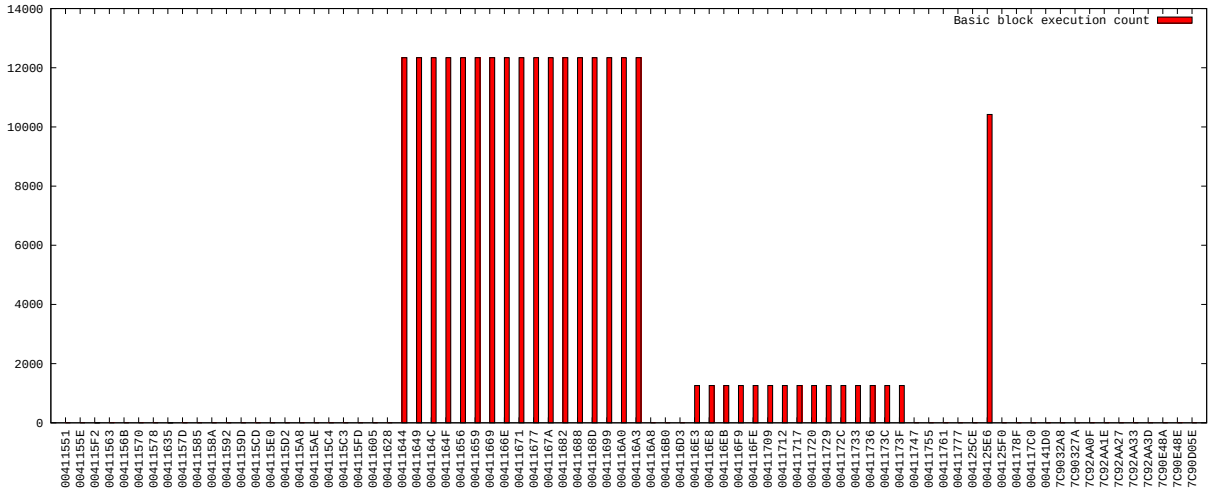

Figure 4: Basic block execution frequency histogram.

```
loop: lods byte ptr ds:[esi]
      rol  al, 0ce
      add  al, cl
      rol  al, 58
      add  al, cl
      ror  al, 3c
      ror  al, 0ed
      sub  al, cl
      ror  al, 5c
      sub  al, 0f3
      ror  al, 40
      dec  al
      sub  al, 48
      ror  al, 2b
      dec  al
      xor  al, 93
      dec  al
      sub  al, 17
      add  al, 0a2
      stos byte ptr es:[edi]
      loopd loop
```

```
loop: lods byte ptr ds:[esi]
      dec  al
      xor  al, 56
      sub  al, 2f
      xor  al, 4e
      ror  al, 9c
      add  al, 22
      dec  al
      rol  al, 0e9
      rol  al, 90
      ror  al, 0a3
      ror  al, 3c
      dec  al
      dec  al
      sub  al, 17
      xor  al, 12
      ror  al, 2e
      sub  al, cl
      add  al, 1c
      add  al, cl
      dec al
      stos byte ptr es:[edi]
      loopd loop
```

Table 1: First two decryption loops of Yoda's protector 1.03.2.

5.3 Limitations

Like MmmBop, this unpacker modifies the address space of the target process. Some packers, especially the ones written by malware authors, might try to detect foreign modules. Also the packer could use hook detection methods to check the integrity of a critical functions, such as `ntdll!NtContinue`. This is not very easy to implement, mostly it depends on how the hook was installed. For instance, if we overwrite the beginning of a function with a jump instruction, the packer can just simply compare the first byte of the function to `E9` which stands for a relative jump opcode. Also, the malware packer could overwrite `kernel32!BaseThreadStartThunk` with its own code that essentially would overwrite unpacker's installed hook. Thus, defeating the unpacker since the unpacker must always retain the control over the packed application.

5.4 Future work

Even though the unpacker can perform simple program traces there is a lot of space for improvements. The

Before decryption				After decryption			
004116a8	59	pop	ecx	004116a8	e803000000	call	004116b0
004116a9	cdb9	int	0B9h	004116ad	eb01	jmp	004116b0
004116ab	b9b9edb555	mov	ecx,55B5EDB9h	004116af	e933dbb92f	jmp	2ffaf1e7
004116b0	8a2d94fe8fce	mov	ch,ds:[0CE8FFE94h]	004116b4	7342	jae	004116f8
004116b6	b9b755defb	mov	ecx,0FBDE55B7h	004116b6	0081e9466e42	add	[ecx+426E46E9h],al
004116bb	ce	into		004116bc	008bd581c246	add	[ebx+46C281D5h],cl
004116bc	b96c05b7d0	mov	ecx,0D0B7056Ch	004116c2	6e	outs	dx,byte ptr [esi]
004116c1	defb	fdivp	st(3),st	004116c3	42	inc	edx
004116c3	ce	into		004116c4	008d3a8bf733	add	[ebp+33F78B3Ah],cl
004116c4	b964ae6c9d	mov	ecx,9D6CAE64h	004116ca	c0e803	shr	al,3
004116c9	8ab859cdb9b9	mov	bh,[eax-464632A7h]	004116cd	0000	add	byte ptr [eax],al
004116cf	b9edb5596c	mov	ecx,6C59B5EDh	004116cf	00eb	add	bl,ch
004116d4	05b3d0d487	add	eax,87D4D0B3h	004116d1	01e8	add	eax,ebp
004116d9	ce	into		004116d3	8bd5	mov	edx,ebp
004116da	b964d177cc	mov	ecx,0CC77D164h	004116d5	81c2c9744200	add	edx,4274C9h
004116df	78ed	js	004116ce	004116db	8d02	lea	eax,[edx]
				004116dd	50	push	eax
				004116de	c3	ret	
				004116df	90	nop	

Table 2: A hidden layer before and after decryption.

unpacker still requires implementation of the basic block analyzer. Also the signature database must be created. Some compiler signatures could be taken from PEiD. However, the signature database that is available with PEiD does not include the signatures of a more recently released compilers. For new compilers signatures would have to be created manually. The unpacker also cannot handle multi-threaded unpacking algorithms, therefore the support for such algorithms must be added. Finally, the unpacker does not carry any defensive mechanism to avoid its detection, thus the defenses should be added too. A possible solution would be to create a kernel driver to protect the unpacker's memory.

6 Commercial value

The commercial value of an automated unpacking solutions can be considered twofold. On one hand such solutions are constantly used by an Anti-Virus software to increase virus detection rate. There is a huge competition in the industry of the Anti-Virus software where the best Anti-Virus product is determined by the best virus detection rate. On the other hand the automated unpacking solutions can target security researchers, in particular malicious software analysts. Even if the market is not very big the demand for such tools is very high. IDA Pro is a good example of targeting a small niche. It is widely used in malicious software analysis and can be extended via plug-ins (e.g. Hex-Rays [5] decompiler).

7 Conclusion

Executable packing can be used for both legitimate purposes and malicious. In this paper we presented a survey of packers by looking at the employed defensive and cryptographic solutions. Additionally a survey of the generic unpackers was presented too by classifying the unpacking methods. As we have seen, there are many generic unpackers, yet most of them are unable to deal with more sophisticated packers or have a huge performance penalty. Finally, we presented a possible generic unpacking alternative. The alternative method might be a better way to unpack executables due to the code execution natively meaning that we do not have a high performance loss. Also, it was designed to automatically identify OEP and penetrate multiple packing layers by searching for compiler signature matches. Additionally, the performance can be increased by combining signature matching with statistical algorithms.

Like most of the existing generic unpacking methods, the one presented here is not perfect. The unpacking and OEP finding methods must be constantly updated in order to handle protections carried by new packers. Also the unpacker can be evaded by performing an in depth memory analysis.

8 Acknowledgements

I would like to thank my project advisor, professor Sven Dietrich, for all his help, advices and great pointers to reading material. Additional thanks go to John Hernandez for his sincere answers and advices.

References

[1] ASPack and ASProtect. http://www.aspack.com/.

[2] Burneye. http://packetstormsecurity.org/groups/teso/indexsize.html.

[3] Exe Protector. http://www.sriharish.com/.

[4] Executable and Linkable Format. http://www.skyfree.org/linux/references/ELF_Format.pdf.

[5] Hex-Rays. http://www.hex-rays.com/.

[6] IDA Pro disassembler. http://www.hex-rays.com/idapro/.

[7] MPRESS. http://www.matcode.com/.

[8] PEiD. http://www.peid.info/.

[9] PELock. http://www.pelock.com/.

[10] Shiva. http://www.securereality.com.au/.

[11] Themida. http://www.oreans.com/.

[12] UPX: the Ultimate Packer for Executables. http://upx.sourceforge.net/.

[13] VMProtect. http://www.vmprotect.ru/.

[14] Windows Cryptography API. http://msdn.microsoft.com/en-us/library/aa380256(VS.85).aspx.

[15] Windows Tools for Debugging. http://www.microsoft.com/whdc/DevTools/Debugging/default.mspx.

[16] Xen Hypervisor. http://www.xen.org/.

[17] Yoda's protector. http://yodap.sourceforge.net/.

[18] A Stream Cipher Encryption Algorithm "Arcfour". http://tools.ietf.org/id/draft-kaukonen-cipher-arcfour-03.txt, 1999.

[19] Advanced Encryption Standard (AES). *Federal Information Processing Standards*, Publication 197, 2001.

[20] Ross Anderson, Eli Biham, and Lars Knudsen. Serpent: A Proposal for the Advanced Encryption Standard. 1998.

[21] AndreaGeddon. Armadillo 4.20: Removing the armour: a naked animal. http://www.reteam.org/papers/e72.pdf, October 2005.

[22] Piotr Bania. Generic Unpacking of Self-modifying, Aggressive, Packed Programs. 2009.

[23] Carolynn Burwick, Don Coppersmith, Edward D'Avignon, Rosario Gennaro, Shai Halevi, Charanjit Jutla, Stephen M. Matyas Jr., Luke O'Connor, Mohammad Peyravian, David Safford, and Nevenko Zunic. MARS—a candidate cipher for AES, September 1999.

[24] Mihai Christodorescu, Johannes Kinder, Somesh Jha, Stefan Katzenbeisser, and Helmut Veith. Malware Normalization. Technical Report 1539, University of Wisconsin, Madison, Wisconsin, USA, November 2005.

[25] Christian Collberg, Clark Thomborson, and Douglas Low. A Taxonomy of Obfuscating Transformations. Technical Report 148, The University of Auckland, Private Bag 92019, Auckland, New Zealend, July 1997.

[26] Microsoft Corporation. Microsoft Portable Executable and Common Object File Format Specification.

[27] Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. Ether: Malware Analysis via Hardware Virtualization Extensions, 2008.

[28] Chris Eagle. Attacking Obfuscated Code with IDA Pro, 2004.

[29] Tim Ebringer, Li Sun, and Serdar Bozdas. A fast randomness test that preserves local detail. In *Virus Bulletin 2008*, 2008.

[30] Eldad Eilam. *Reversing: Secrets of Reverse Engineering*. Wiley Publishing, 2005.

[31] Peter Ferrie. Anti-Unpacker Tricks. In *Proceedings of the 2nd International CARO Workshop*, 2008.

[32] Ken Johnson. A catalog of NTDLL kernel mode to user mode callbacks, part 2: KiUserExceptionDispatcher. http://www.nynaeve.net/?p=201, November 2007.

[33] Ken Johnson. A catalog of NTDLL kernel mode to user mode callbacks, part 3: KiUserApcDispatcher. http://www.nynaeve.net/?p=202, November 2007.

[34] Min Gyung Kang, Pongsin Poosankam, and Heng Yin. Renovo: A Hidden Code Extractor for Packed Executables. In *Proceedings of the 5th ACM Workshop on Recurring Malcode (WORM'07)*, 2007.

[35] E. Labir. Unpacking by Code Injection. *Code Breakers journal*, 1(2), 2004.

[36] Lorenzo Martignoni, Mihai Christodorescu, and Somesh Jha. OmniUnpack: Fast, Generic and Safe Unpacking of Malware. In *ACSAC*, 2007.

[37] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.

[38] Gary Nebbett. *Windows NT/2000 Native API Reference*. New Riders Publishing, Thousand Oaks, CA, USA, 2000.

[39] Markus F.X.J. Oberhumer. LZO real-time data compression library. `http://www.oberhumer.com/opensource/lzo/`.

[40] Markus F.X.J. Oberhumer. UCL data compression library. `http://www.oberhumer.com/opensource/ucl/`.

[41] Dirk Paehl. SFX Creator. `http://www.paehl.de/cms/sfx_creator`.

[42] Phillip Porras, Hassen Saidi, and Vinod Yegneswaran. A Foray into Conficker's Logic and Rendezvous Points. In *2nd USENIX Workshop on Large-scale Exploits and Emergent Threats (LEET '09)*, 2009.

[43] Silicon Realms. Software Passport. `http://www.siliconrealms.com/software-passport-armadillo.html`.

[44] R.L. Rivest, M.J.B. Robshaw, R.Sidney, and Y.L. Yin. The RC6 Block Cipher. 1998.

[45] Ronald L. Rivest. The MD5 Message-Digest Algorithm (RFC 1321). `http://www.ietf.org/rfc/rfc1321.txt`, April 1992.

[46] Paul Royal, Mitch Halpin, David Dagon, Robert Edmonds, and Wenke Lee. PolyUnpack: Automating the Hidden-Code Extraction of Unpack Executing Malware. In *ACSAC '06: Proceedings of the 22nd Annual Computer Security Application Conference on Annual Computer Security Application Conference*, pages 289–300, Washington, DC, USA, 2006. IEEE Computer Society.

[47] B. Schneier. Description of a New Variable-Length Key, 64-Bit Block Cipher (Blowfish). In *Fast Software Encryption, Cambridge Security Workshop Proceedings*, pages 191–204. Springer-Verlag, 1994.

[48] B. Schneier, J. Kelsey, D. Whiting, D. Wagner, C. Hall, and N. Ferguson. Twofish: A 128-Bit Block Cipher. 1998.

[49] Monirul Sharif, Andrea Lanzi, Jonathon Giffin, and Wenke Lee. Automatic Reverse Engineering of Malware Emulators. In *Proceedings of The 2009 IEEE Symposium on Security and Privacy (Oakland 09)*, Oakland, CA, 2009.

[50] skape. Using dual-mappings to evade automated unpackers. `http://uninformed.org/?v=10&a=1`.

[51] Joe Stewart. OllyBonE v0.1, Break-On-Execute for OllyDbg. `http://www.joestewart.org/ollybone`.

[52] Li Sun, Tim Ebringer, and Serdar Bozdas. Hump-and-dump: efficient generic unpacking using an ordered address execution histogram. In *Second International CARO Workshop*, Hoofddorp, Netherlands, 2008.

[53] Peter Szor. *The Art of Computer Virus Research and Defense*. Addison-Wesley Professional, February 2005.

[54] Wikipedia. Win32 Thread Information Block. `http://en.wikipedia.org/wiki/Win32_Thread_Information_Block`.

[55] Zhichen Xu, Barton P. Miller, and Oscar Naim. Dynamic instrumentation of threaded applications. *SIGPLAN Not.*, 34(8):49–59, 1999.

[56] Oleh Yuschuk. OllyDbg. `http://www.ollydbg.de/`.

A IDA Pro and OllyDbg

IDA Pro and OllyDbg are two very well known debuggers that are widely used in reverse engineering and malicious software analysis. Both of them provide a plug-in API that can be used to extend the functionality of these debuggers. An unpacking method similar to the one presented in the paper could also be implemented as a plug-in for the debugger. The main disadvantage of creating a plug-in in this case is that the analysis target is being debugged. All Windows debuggers use Windows Debugging API that allow to manipulate the

debuggee. Additionally it enables certain flags in the debugged application and simply tells the application that it is being debugged. In some cases the presence of the debugger can be hidden with memory modifications from the user-mode. In other cases, this might not be possible and would require a kernel driver to hide the debugger. This was the main reason why the unpacker was not implemented as a debugger plug-in. By implementing the unpacker as a standalone application one can achieve more transparency and avoid being detected with anti-debugger tricks.

B Step-by-step decryption

This section shows first few iterations of the first decryption loop used in Yoda's Protector. The decryption loop has 21 steps. Every step is numbered from 1 to 21. The tables below show how data changes when going through decryption loop step-by-step. The hyphens in the table mean that register state at a particular step has not changed.

```
Decryption loop:
1: loop: lods byte ptr ds:[esi]
2:       rol  al, 0ce
3:       add  al, cl
4:       rol  al, 58
5:       add  al, cl
6:       ror  al, 3c
7:       ror  al, 0ed
8:       sub  al, cl
9:       ror  al, 5c
10:      sub  al, 0f3
11:      ror  al, 40
12:      dec  al
13:      sub  al, 48
14:      ror  al, 2b
15:      dec  al
16:      xor  al, 93
17:      dec  al
18:      sub  al, 17
19:      add  al, 0a2
20:      stos byte ptr es:[edi]
21:      loopd loop

Initial CPU state:
EAX = 0, ECX = 3031, ESI = EDI = 4116AB
Data starting
    at 4116A8: 59 CD B9 B9 B9 ED B5 ...
```

```
Iteration #1
    1      2  3  4  5  6  7  8  9  10 11 12 13 15 15 16 17 18 19 20      21
AL  59     56 87 87 B8 8B 5C 2B B2 BF BF BE 76 CE CD 5E 5D 46 E8 -       -
ECX 3031   -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -       3030
ESI 4116A9 -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -       -
EDI 4116A8 -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  4116A9  -
Data at 4116A8: E8 CD B9 B9 B9 ED B5 ...

Iteration #2
    1      2  3  4  5  6  7  8  9  10 11 12 13 15 15 16 17 18 19 20      21
AL  CD     73 A3 A3 D3 3D E9 B9 9B A8 A8 A7 5F EB EA 79 78 61 03 -       -
ECX 3030   -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -       302F
ESI 4116AA -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -       -
EDI 4116A9 -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  4116AA  -
Data at 4116A8: E8 03 B9 B9 B9 ED B5 ...

Iteration #3
    1      2  3  4  5  6  7  8  9  10 11 12 13 15 15 16 17 18 19 20      21
AL  B9     6E 9D 9D CC CC 66 37 73 80 80 7F 37 E6 E5 76 75 5E 00 -       -
ECX 302F   -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -       302E
ESI 4116AB -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -       -
EDI 4116AA -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  4116AB  -
Data at 4116A8: E8 03 00 B9 B9 ED B5 ...

Iteration #4
    1      2  3  4  5  6  7  8  9  10 11 12 13 15 15 16 17 18 19 20      21
AL  B9     6E 9C 9C CA AC 65 37 73 80 80 7F 37 E6 E5 76 75 5E 00 -       -
ECX 302E   -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -       302D
ESI 4116AC -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -       -
EDI 4116AB -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  4116AC  -
Data at 4116A8: E8 03 00 00 B9 ED B5 ...

Iteration #5
    1      2  3  4  5  6  7  8  9  10 11 12 13 15 15 16 17 18 19 20      21
AL  B9     6E 9B 9B C8 8C 64 37 73 80 80 7F 37 E6 E5 76 75 5E 00 -       -
ECX 302D   -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -       302C
ESI 4116AD -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -       -
EDI 4116AC -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  4116AD  -
Data at 4116A8: E8 03 00 00 00 ED B5 ...

Iteration #6
    1      2  3  4  5  6  7  8  9  10 11 12 13 15 15 16 17 18 19 20      21
AL  ED     7B A7 A7 D3 3D E9 BD DB E8 E8 E7 9F F3 F2 61 60 49 EB -       -
ECX 302C   -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -       302B
ESI 4116AE -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -       -
EDI 4116AD -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  4116AE  -
Data at 4116A8: E8 03 00 00 00 EB B5 ...

Iteration #7
    1      2  3  4  5  6  7  8  9  10 11 12 13 15 15 16 17 18 19 20      21
AL  B5     6D 98 98 C3 3C E1 B6 6B 78 78 77 2F E5 E4 77 76 5F 01 -       -
ECX 302B   -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -       302A
ESI 4116AF -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -       -
EDI 4116AE -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  -  4116AF  -
```

```
Data at 4116A8: E8 03 00 00 00 EB 01 ...
```

By looking at the data above one can clearly see that the decryption loop was generated by a polymorphic code engine and it is not a compiler generated code. First of all, compilers usually optimize code, however when we look at the generated data we see that the step 4 is redundant. Also the steps 17, 18 and 19 can be merged into one `ADD` or `SUB` instruction. The reason behind such code is that it provides different signatures. The algorithm itself does not change but the code of the algorithm is different.