



Reverse Engineering Malware Binary Obfuscation and Protection

Jarkko Turkulainen

F-Secure Corporation

Protecting the irreplaceable | [f-secure.com](https://www.f-secure.com)



Binary Obfuscation and Protection

What is covered in this presentation:

- Runtime packers
- Compression algorithms
- Packer identification
- Unpacking strategies
- Unpacking examples on simple systems
- Custom protection systems

Java and JavaScript shrinkers and obfuscators are not covered here!

Overview of runtime packers

- Runtime packer combines a compressed executable file with a decompressor in a single executable file
- Packers are used to shrink the size of executables
- Because the data is compressed, it usually not clear-text, also acting as protective layer
- Packers are also used for protecting executables against debugging, dumping and disassembling
- Most modern malware use some sort of runtime packer
- If static analysis of malware is needed, protective layer(s) must be opened
- Tens of different runtime packers easily available
- Some advanced systems are commercial

Compression algorithms

- Statistical
 - Data symbols are replaced with symbols requiring smaller amount of data
 - Common symbols are presented with fewer bits than less common ones
 - Symbol table is included with the data
 - Example: Huffman coding
- Dictionary-based
 - Data symbols are stored in a dictionary
 - Compressed data references to the dictionary
 - Static: dictionary included with the data
 - Sliding window: dictionary is based on previously seen input data
 - Example: LZ

Common packers

- UPX (Ultimate Packer for eXecutables). Simple runtime packer. Supports multiple target platforms. Compression algorithms: UCL, LZMA (both LZ-based dictionary models)
- FSG: Simple packer for Win32. Compression: aplib (LZ-based)
- MEW: Simple packer for Win32 (aplib)
- NSPACK: Simple packer for Win32 (LZMA)
- UPACK: Simple packer for Win32 (aplib)

Simple packers

- Most common packers are very simple (UPX, FSG etc.)
- Single-process, (usually) single-thread
- Single-layer compression/encryption
- Might use some trivial anti-debug tricks
- Doesn't modify the code itself (works at link-level)
- Implementation not necessarily simple!

Complex packers

- Uses multiple processes and threads
- Multi-layer encryption (page, routine, block)
- Advanced anti-debugging techniques
- Code abstraction (metamorphic, virtual machines etc.)
- Examples: Armadillo, Sdprotect, ExeCrypt, VMProtect

Packer platforms

- Almost all packers run on Windows and DOS
- UPX is a notable exception (Linux, OSX, BSD, different CPU platforms)
- Android:
 - UPX supports Linux/ARM, so at least in theory Android native shared libraries could be packed
 - OT: Classes in DEX files can be packed with Java packers and then converted to Dalvik

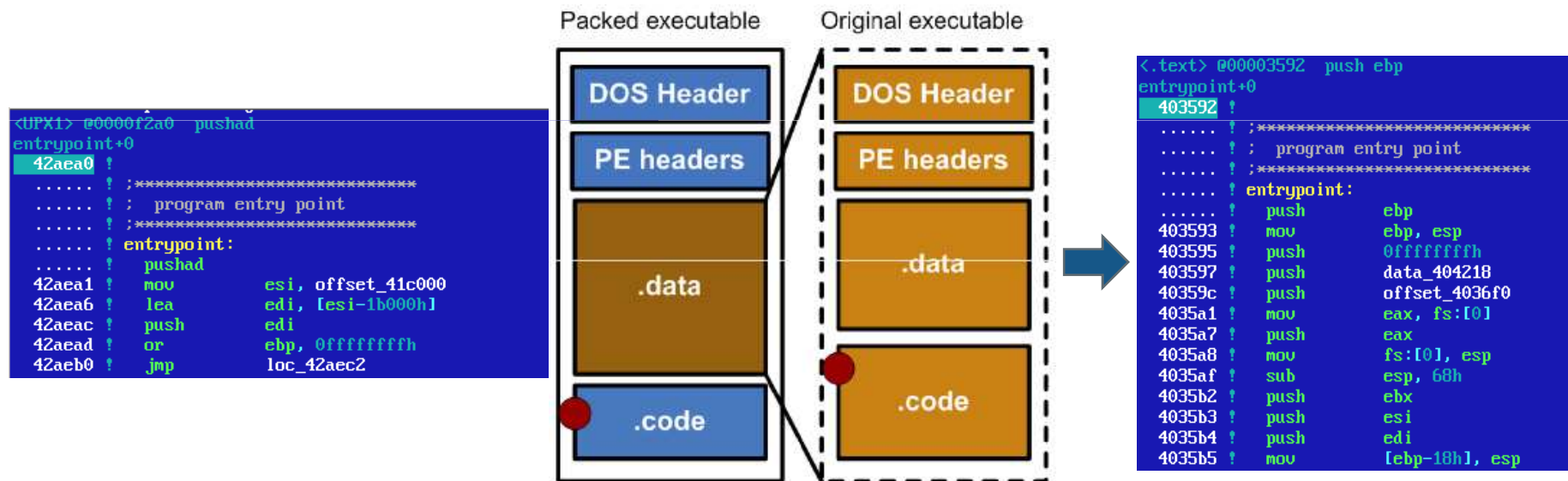
Anatomy of typical packed file

- Weird PE section names
- Sections are very dense (high Shannon's entropy)
- Small amount of imported functions
- Entry code looks bogus

(HT Demo)

How typical packer runtime works

1. Original data is located somewhere in the packer code data section
2. Original data is uncompressed to the originally linked location
3. Control is transferred to original code entry point (OEP)



Anti-* tricks

- Complex packers utilize lots of tricks to fool debuggers, disassemblers, dumpers etc.
- Example anti-debugging trick: debug-bit in PEB (Windows API: IsDebuggerPresent)
- For more details, see lecture slides “Dynamic Analysis I”

(PEB demo)

How to identify packers

- Known characteristics of PE envelope (section names, entry point code etc.)
- PE identification utilities (for example: PEiD)
- Not foolproof!

```
* PE header at offset 000000f0
[+] COFF header
[+] optional header
[+] optional header: NT fields
[+] optional header: directories
[+] section header 0: UPX0 rva 00001000 vsize 0001b000
[+] section header 1: UPX1 rva 0001c000 vsize 0000f000
[+] section header 2: .rsrc rva 0002b000 vsize 00001000
```



```
00000000 4d 5a 50 00 02 00 00 00-04 00 0f 00 ff ff 00 00 |MZP
00000010 b8 00 00 00 00 00 00 00-40 00 1a 00 00 00 00 00 |
00000020 00 00 00 00 00 00 00 00-00 00 00 00 00 00 00 00 |
00000030 00 00 00 00 00 00 00 00-00 00 00 00 40 00 00 00 |
00000040 50 45 00 00 4c 01 02 00-46 53 47 21 00 00 00 00 |PE L FSG!
00000050 00 00 00 00 e0 00 8e 81-0b 01 00 00 00 38 00 00 |
00000060 00 12 00 00 00 00 00 00-45 eb 00 00 00 10 00 00 |
00000070 00 50 00 00 00 00 40 00-00 10 00 00 00 02 00 00 |P
00000080 01 00 00 00 00 00 00 00-04 00 00 00 00 00 00 00 |
```

Unpacking strategies

- Static unpacking
 - Unpacking without actually running the file
 - Algorithm-specific
 - Very difficult and time-consuming to implement
 - Fast, reliable
 - System-independent
- Dynamic (live) unpacking
 - Generic
 - Low-cost, easy to implement
 - Needs to be run on native platform
- Combined approach (emulators)
 - Flexibility of dynamic unpacking + security of static unpacking
 - Extremely hard to implement

Static unpacking

- Requires knowledge about the routines and algorithms used by the packer
- Unpacking is basically just static implementation of the work done by unpacker stub when the file is run:
 - Locate the original data
 - Uncompress and/or decrypt the data
 - Fix imports, exports, resources etc. data structures
- Some packers include unpacker that can completely restore the original file (well, at least UPX has it with `-d` option)
- The file is not run - secure and fast

(UPX + PEID demo)

Dynamic unpacking

- Idea: let the program run on a real system and unpack itself
- Needs isolated, real machine (VMWare might not be good enough!)
- Basic tools are freely available (hex editors, debuggers etc.)

Dynamic unpacking with debugger

- Packed file is opened with debugger, or debugger is attached to already running target
- Let the packer stub run and unpack the original program
- Save the unpacked data to disk or analyze using tools provided by the debugger
- Problems with debugger:
 - Debugger detection (PEB debug bit, anti-debug tricks etc.)
 - Debugger attacks (throwing exceptions etc.)

Dynamic unpacking with dumping

- Run the file
- Dump the process memory on disk, pseudo code:

```
void Dump(DWORD pid)
{
    BYTE buf[PAGE_SIZE];
    DWORD address, written;
    HANDLE hFile = CreateFile("dump.dat", GENERIC_WRITE,
        0, NULL, CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
    HANDLE hProcess = OpenProcess(PROCESS_VM_READ, FALSE, pid);
    for (address = 0; address < 0x80000000; address += PAGE_SIZE)
    {
        if (ReadProcessMemory(hProcess, (LPVOID)address, buf, PAGE_SIZE, NULL))
        {
            writeFile(hFile, buf, PAGE_SIZE, &written, NULL);
        }
    }
}
```

PE reconstruction

- Dumped image is more usable if it can be opened with RE tools like IDA
- PE envelope needs to be build around the dumped image:
 - The image can be mapped as a single section
 - Original Entry Point (OEP) needs to be figured out
 - Import Address Table (IAT) needs to be reconstructed
- IAT reconstruction can cause lot of problems:
 - Packers build IAT dynamically
 - IAT entries may not be direct addresses to the imported function, it can be some kind of trampoline
- OEP can be tricky to find
- Tools like ImpRec and OllyDump can automate the reconstruction process

Examples: unpacking simple packers

- Try to identify the packer based on PE characteristics
- Use static unpacking tools (if available)
- Use dynamic methods (OllyDbg/Immunity)

(Demo)

Example unpacking tool: FIST

- FIST is a proprietary tool for generic unpacking
- Based on hooking Win32 function calls:
 - Code in the return address of Win32 call is compared to the disk image
 - If code is not on disk, it is most likely dynamically generated
 - OEP can be found by tracing back to known function prolog signatures
 - Note that disk image needs to be mapped to virtual addresses (most simple way to do this is to execute an instance of the image as suspended and use that as a base disk image)

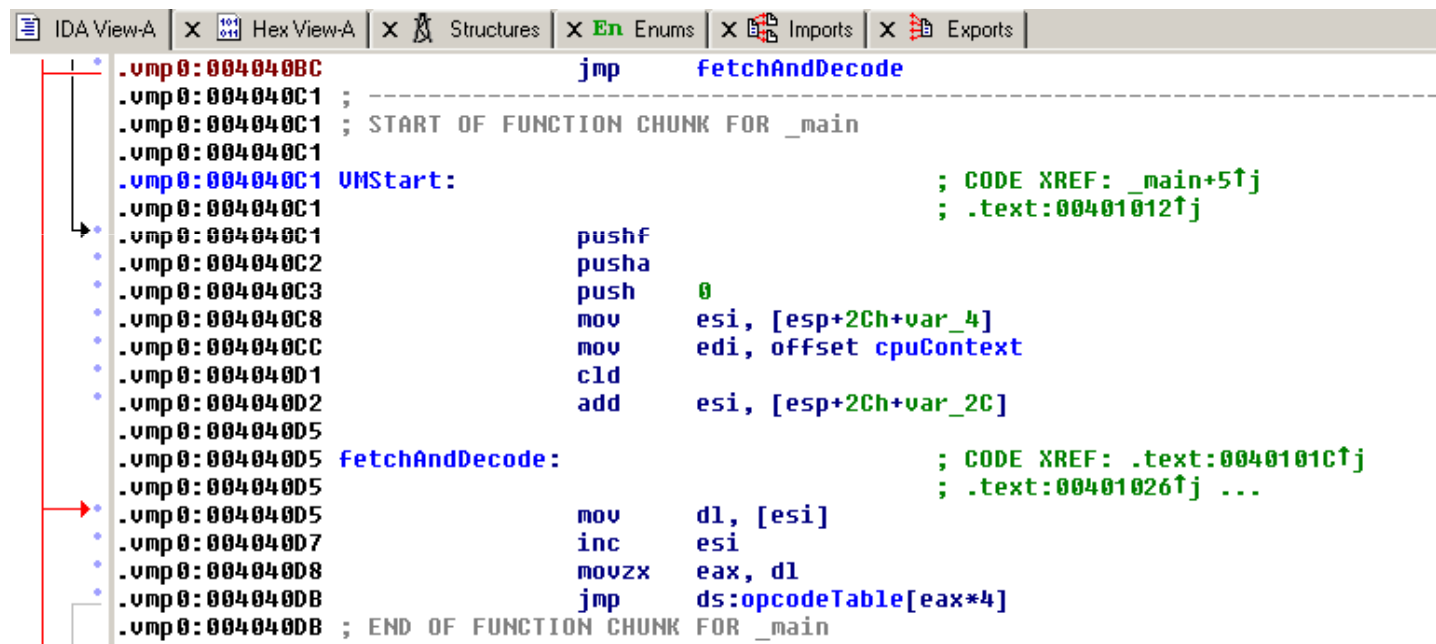
Demo: Unpacking example files with FIST

If this looks too simple...

- Live unpacking of simple envelopes is easy, BUT...
- Imports are usually lost in the unpacking process
- Debuggers are often very unreliable, they can be detected (even when attaching!)
- Complex protection systems are becoming more popular
- Malware can also use “custom protection systems”

Complex protection system example: VMProtect

- Protects selected parts of the program with virtual machine
- Also has additional layers of protection: obfuscation, anti-debugging etc.



```
IDA View-A | Hex View-A | Structures | En Enums | Imports | Exports
.vmp0:004040BC jmp fetchAndDecode
.vmp0:004040C1 ; -----
.vmp0:004040C1 ; START OF FUNCTION CHUNK FOR _main
.vmp0:004040C1
.vmp0:004040C1 VMStart: ; CODE XREF: _main+51j
.vmp0:004040C1 ; .text:004010121j
.vmp0:004040C1 pushf
.vmp0:004040C2 pusha
.vmp0:004040C3 push 0
.vmp0:004040C8 mov esi, [esp+2Ch+var_4]
.vmp0:004040CC mov edi, offset cpuContext
.vmp0:004040D1 cld
.vmp0:004040D2 add esi, [esp+2Ch+var_2C]
.vmp0:004040D5
.vmp0:004040D5 fetchAndDecode: ; CODE XREF: .text:0040101C1j
.vmp0:004040D5 ; .text:004010261j ...
.vmp0:004040D5 mov dl, [esi]
.vmp0:004040D7 inc esi
.vmp0:004040D8 movzx eax, dl
.vmp0:004040DB jmp ds:opcodeTable[eax*4]
.vmp0:004040DB ; END OF FUNCTION CHUNK FOR _main
```

(Demo)

Custom protections systems

- Usually works at compiler-level (integrates with the source code)
- Most common case is data encryption with some simple algorithm, like bit-wise ADD/XOR/etc.
- Sometimes a bit heavier toolset is required: IDA, IDAPython (python scripting for IDA)
- Live unpacking with debuggers might also solve some custom system cases as well!

Example custom system: Bobic worm string encryption

```

File Edit Jump Search View Debug Options Window
[.] IDA View-A
UPX0:1000BCAB loc_1000BCAB: ; CODE XREF: sub_1000BC59+30^j
UPX0:1000BCAB lea eax, [ebp-0B0h]
UPX0:1000BCB1 push eax
UPX0:1000BCB2 mov dword ptr [ebp+10h], offset aWhHae7LuhoIVsc ; '*\WH\h`aE7+lvHo+I+usch1*6`yeL:
UPX0:1000BCB9 mov dword ptr [ebp+14h], offset a@vSp_xRU17ev6o ; "@v\sp.x<R;U17Ev>!\`60bFdBFF`?
UPX0:1000BCC0 mov dword ptr [ebp+18h], offset a4k1putEQoa@f_1 ; "<4k1pUt`)E/Qoa@F*_{1yJ1-0Z?=&h
UPX0:1000BCC7 mov dword ptr [ebp+1Ch], offset aY@GaCIffnF9I59 ; ")y@)gA;c-\If+fFN`"f9=i5%l9XgkB
UPX0:1000BCCE call sub_1000C5A8
UPX0:1000B
UPX0:1000B File Edit Jump Search View Debug Options Window
[.] IDA View-A
UPX0:1000B loc_1000BCAB: ; CODE XREF: sub_1000BC59+38^j
UPX0:1000B lea eax, [ebp-0B0h]
UPX0:1000B push eax
UPX0:1000B UPX0:1000BCB1 mov dword ptr [ebp+10h], offset aWhHae7LuhoIVsc ; "download.yahoo.com/dl/insta
UPX0:1000B UPX0:1000BCB9 mov dword ptr [ebp+14h], offset a@vSp_xRU17ev6o ; "ftp.scarlet.be/pub/mozilla
UPX0:1000B UPX0:1000BCC0 mov dword ptr [ebp+18h], offset a4k1putEQoa@f_1 ; "ftp.newaol.com/aim/win95/I
UPX0:1000B UPX0:1000BCC7 mov dword ptr [ebp+1Ch], offset aY@GaCIffnF9I59 ; "g.msn.com/?MEEN_US/EN/SETU
UPX0:1000B UPX0:1000BCCE call sub_1000C5A8
UPX0:1000B UPX0:1000BCD3 push 4
UPX0:1000B UPX0:1000BCD5 pop ecx
UPX0:1000B UPX0:1000BCD6 cdq
UPX0:1000B UPX0:1000BCD7 idiv ecx
UPX0:1000BCD9 lea ecx, [ebp-0A8h]
UPX0:1000BCDF push dword ptr [ebp+edx*4+10h]
UPX0:1000BCE3 call sub_1000CBE7
UPX0:1000BCEB mov ecx, eax
UPX0:1000BCEA call decrypt
UPX0:1000BCEF push eax
UPX0:1000BCF0 lea eax, [ebp+48h]
UPX0:1000BCF3 push offset aHttp ; "http://"
UPX0:1000BCF8 push eax
  
```


Conclusions

- Live unpacking is easy and cost-effective way to handle most malware
- For handling complex protection systems, custom decryptors, tracers and memory dumpers must be implemented

Thanks for your patience!

Further reading

- Wikipedia on runtime packing - http://en.wikipedia.org/wiki/Executable_compression
- UPX - <http://upx.sourceforge.net/>
- IDAPython - <http://d-dome.net/idapython>
- “Runtime Packers: The Hidden Problem?” - <http://www.blackhat.com/presentations/bh-usa-06/BH-US-06-Morgenstern.pdf>
- “The Art of Unpacking” - <https://www.blackhat.com/presentations/bh-usa-07/Yason/Presentation/bh-usa-07-yason.pdf>
- Bobic worm description: http://www.f-secure.com/v-descs/bobic_k.shtml

Protecting
the
irreplaceable

