Reverse Engineering Malware
Binary Obfuscation and Protection

Jarkko Turkulainen

F-Secure Corporation

Binary Obfuscation and Protection

What is covered in this presentation:

- Runtime packers
- Compression algorithms
- Packer identification
- How to unpack
- Unpacking examples on simple systems
- Custom protection systems

Java/DEX and JavaScript shrinkers and obfuscators are not covered here!

F-Secure.

Overview of runtime packers

- Original purpose: reduce size on disk
- Runtime packer combines a compressed executable file with a decompressor in a single executable file
- Usually decompression is done in-memory
- 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 using offset and length
  - 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 source 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

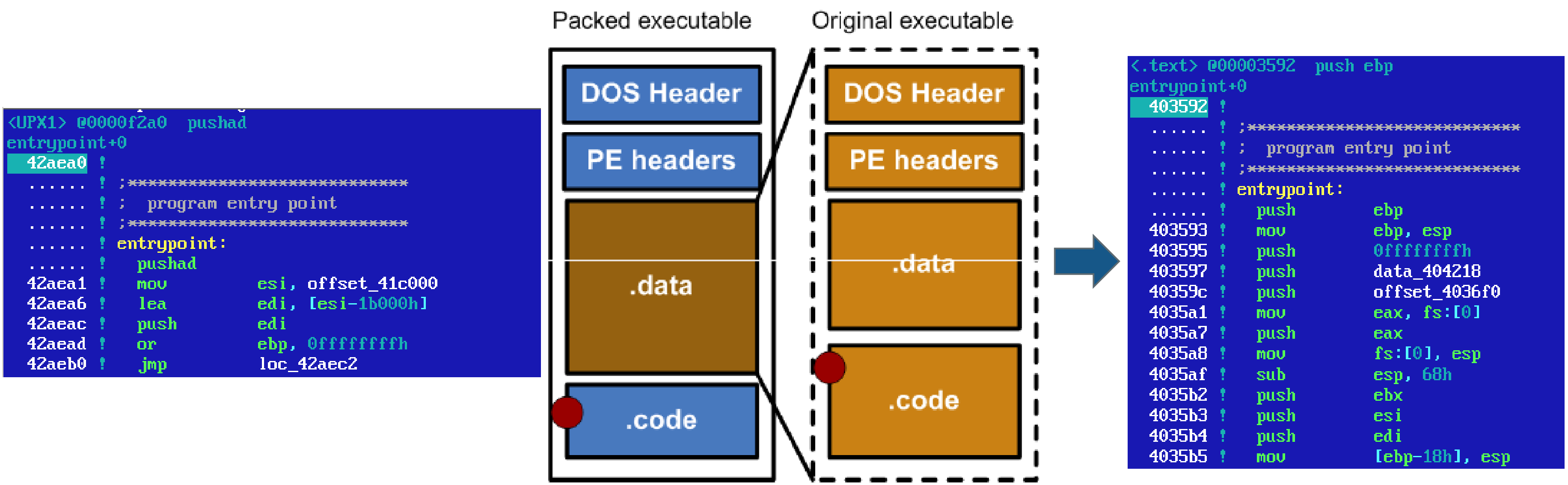
Anatomy of typical packed file

- 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)



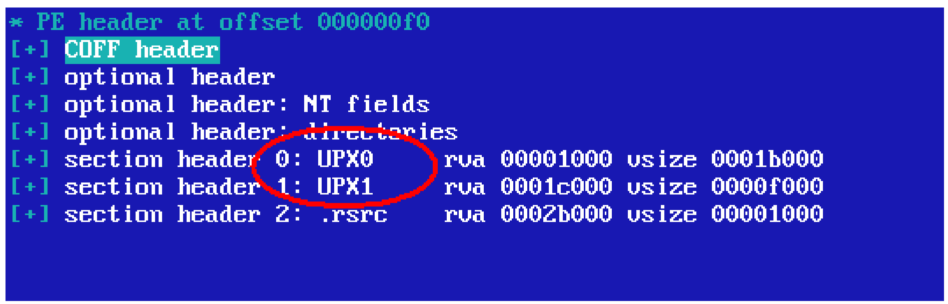
Anti-* tricks

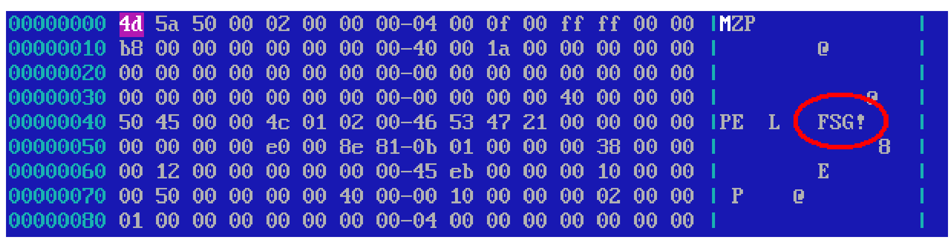
- Complex packers utilize lots of tricks to fool debuggers, disassemblers, dumpers and emulators
- Example anti-debugging trick: debug-bit in PEB (Windows API: IsDebuggerPresent)
- For more details, see lecture slides “Dynamic Analysis”

(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!





How to unpack

- Statically
  - Unpacking without actually running the file
  - Algorithm-specific
  - Very difficult and time-consuming to implement
  - Fast, reliable
  - System-independent
- Dynamically
  - 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
- How to get control:
  - Set breakpoints in known Win32 API's
  - Just run and let the program handle exceptions
  - Break in if program continues running
- 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.)

Debugger automation with scripting

- Debuggers can be extended with flexible scripting languages like python
- Any debugging task can be automated: unpacking, decrypting strings, etc.
- Debuggers that support python scripting:
  - Immunity debugger
  - Python plugin available for OllyDbg
  - GDB
  - IDA debugger
- Other scripting languages
  - Windbg scripting
  - OllyDbg scripting plugins (several languages)
- Python debugger module for Windows:
  - PaiMei, reverse engineering framework includes ”PyDbg” module
  - F-secure proprietary python Win32 debugger using ctypes

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)

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+5↑j
.vmp0:004040C1                                         ; .text:00401012↑j
.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:0040101C↑j
.vmp0:004040D5                                         ; .text:00401026↑j ...
.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)

F-Secure.

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.
- Using a non-common language, like Visual Basic can also be considered as a “protection system”
- 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!

IDA automation

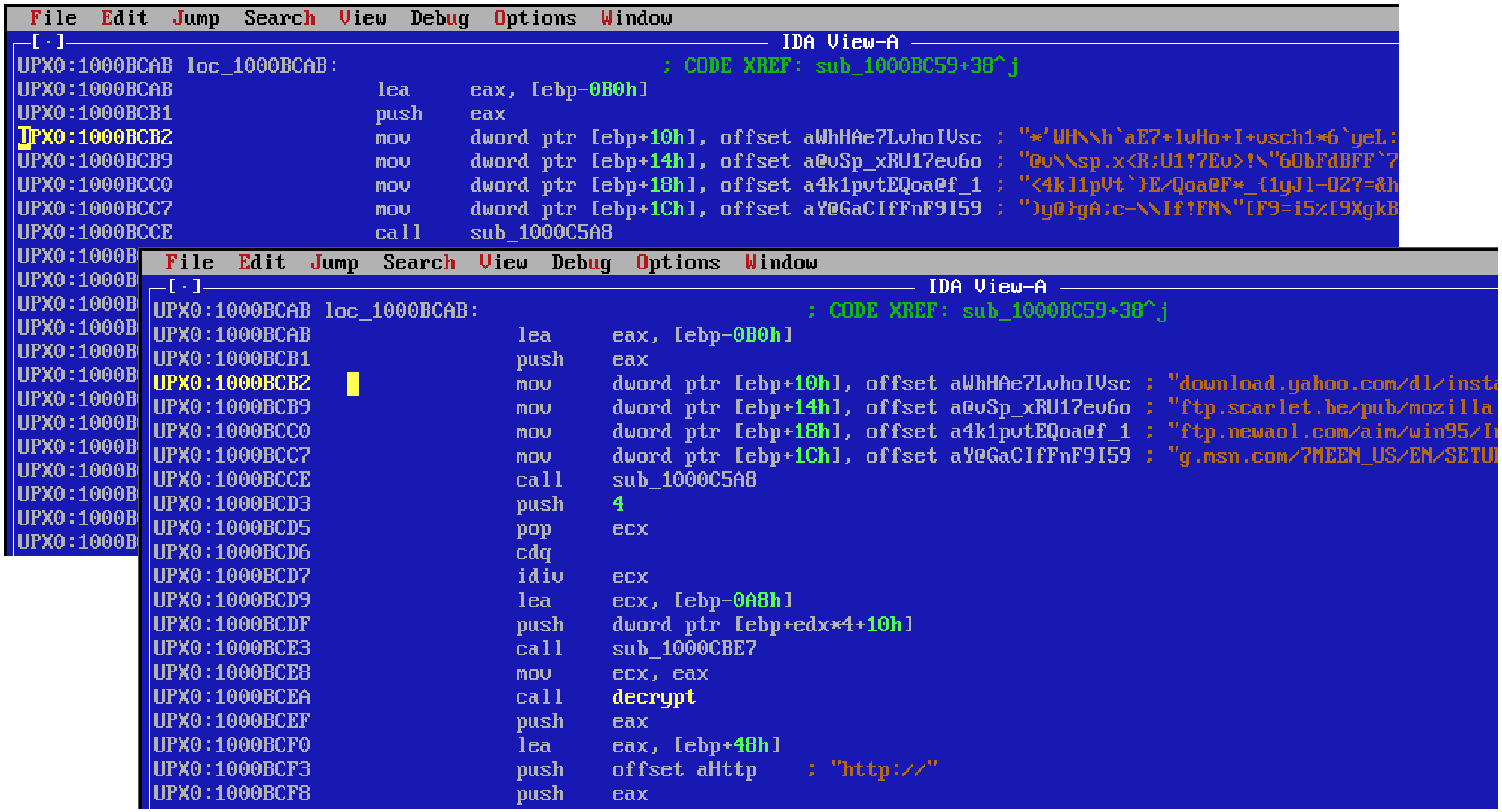
- IDA can be automated with several programming environments:
  - IDA plugin interface (programming language: C/C++)
  - IDC, IDA C-like scripting language
  - IDAPython, python bindings to IDA plugin interface
- Example usage:
  - Reading and modifying the IDA database
  - Renaming functions, commenting
  - Graphing, statistics
  - New processor and loader modules (plugin interface)

Simple decryption loop with IDAC

- Bit-wise XOR over a given address range

```
static decrypt(from, size, key) {
  auto i, x;              // we define the variables
  for ( i=0; i < size; i=i+1 ) {
    x = Byte(from);       // fetch the byte
    x = (x^key);          // decrypt it
    PatchByte(from,x);    // put it back
    from = from + 1;      // next byte
  }
}
```

Example custom system: Bobic worm string encryption



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

F-Secure.