CS 6V81-05: System Security and Malicious Code Analysis
Fighting for Malware: Unpacking, Disassembling, Decompilation

Zhiqiang Lin

Department of Computer Science
University of Texas at Dallas

April 18th, 2012
Outline

1 Background

2 Malware Revese Engineering
   - Unpacking
   - Disassembly
   - Decompilation
   - Program Understanding

3 Summary
Outline

1 Background

2 Malware Reverse Engineering
   - Unpacking
   - Disassembly
   - Decompilation
   - Program Understanding

3 Summary
Background

AV industry in 1998

AV industry in 2008

Image Copyright: DIARUS Security Software GmbH
State-of-the-art in Today’s malware

- Large botnets
- Diverse propagation vectors, exploits, C&C
- Capabilities – backdoor, keylogging, rootkits,
- Logic bombs, time-bombs
- Diverse targets: desktops, mobile platforms, SCADA systems (Stuxnet)

Credit: Most of the slides in this lecture are compiled from Hassen Saidi’s presentation on “Reverse Engineering Malware”
Motivation

- Malware is not about script-kiddies anymore, it’s real business.
- Recent events indicate that it can be a powerful weapon in cyber warfare.
- Manual reverse-engineering is close to impossible
  - Need **automated techniques** to extract system logic, interactions and side-effects, derive intent, and devise mitigating strategies.
Capturing Malware

- Honeynets: Capture malware that scans the Internet for vulnerable targets
- Mining SPAM for attachments
- Mining SPAM for malicious URLs, and capturing drive-by downloads
- AV heuristics
Malware Analysis = Reverse Engineering

Typically a stripped binary with no debugging information.

In the case of malicious code, it is often obfuscated and packed.

Often has embedded suicide logic and anti-analysis logic.

Challenges:
- Lack of automation
- Time-critical analysis
- Labor intensive
- Requires a human in the loop

What does the malware do
How does it do it
Identify triggers
What is the purpose of the malware
Is this an instance of a known threat or a new malware
Who is the author
...
Dynamic vs Static Malware Analysis

Dynamic Analysis
- Techniques that profile actions of binary at runtime
- More popular
  - CWSandbox, TTAnalyze, multipath exploration
  - Only provides partial “effects-oriented profile” of malware potential

Static Analysis
- Can provide complementary insights
- Potential for more comprehensive assessment
Malware Evasions and Obfuscations

- To defeat signature based detection schemes
  - Polymorphism, metamorphism: started appearing in viruses of the 90’s primarily to defeat AV tools

- To defeat Dynamic Malware Analysis
  - Anti-debugging, anti-tracing, anti-memory dumping
  - VMM detection, emulator detection

- To defeat Static Malware analysis
  - Encryption (packing)
  - API and control-flow obfuscations
  - Anti-disassembly

- The main purpose of obfuscation is to slow down the security community
Goal: Systematic and Automatic

- Unpack most of contemporary malware
- Handle most if not all packers
- Deobfuscate API references
- Automate identification of capabilities
- Provide feedback on unpacking success
- Simplify and annotate call graphs to illustrate interactions between key logical blocks
- Enable decompilation of assembly code into a higher-level language
- Identify key logical blocks (crypto, unpacking for instance)
- Reuse certain logical parts
- ...
Reverse Engineering Phases

Phase-I: Unpacking

- Unpacking phase: the image of a running malware sample is often considered damaged:
  - No known OEP. Imported APIs are invoked dynamically and the original import table is destroyed. Arbitrary section names and r/w/e permissions.

Phase-II: Disassembly

- Identification of code and data segments
- Relies on the unpacker to capture all code and data segments.
Reverse Engineering Phases

Phase-III: Decompilation
- Reconstruction of the code segment into a C-like higher level representation
- Relies on the disassembler to recognize function boundaries, targets of call sites, imports, and OEP.

Phase-IV: Program understanding
- Relies on the decompiler to produce readable C code, by recognizing the compiler, calling conventions, stack frames manipulation, functions prologs and epilogs, user-defined data structures.
Outline

1. **Background**

2. **Malware Reverse Engineering**
   - Unpacking
   - Disassembly
   - Decompilation
   - Program Understanding

3. **Summary**
Outline

1 Background

2 Malware Reverse Engineering
   - Unpacking
     - Disassembly
     - Decompilation
     - Program Understanding

3 Summary
Phase 1: Malware Unpacking
What’s the problem

Executable compression

Executable compression is any means of compressing an executable file and combining the compressed data with decompression code into a single executable. When this compressed executable is executed, the decompression code recreates the original code from the compressed code before executing it. In most cases this happens transparently so the compressed executable can be used in exactly the same way as the original.

Code → Data → Code

### Available Packer

<table>
<thead>
<tr>
<th>Name</th>
<th>Latest stable</th>
<th>Software license</th>
<th>x86-64 support</th>
</tr>
</thead>
<tbody>
<tr>
<td>.netshrink</td>
<td>2.3 (March 29, 2012)</td>
<td>Proprietary</td>
<td>Yes</td>
</tr>
<tr>
<td>Armadillo Packer</td>
<td>8.60 (July 6, 2011)</td>
<td>Proprietary</td>
<td>Yes</td>
</tr>
<tr>
<td>ASPack</td>
<td>2.28 (May 18, 2011)</td>
<td>Proprietary</td>
<td>?</td>
</tr>
<tr>
<td>ASPR (ASProtect)</td>
<td>1.64 (September 1, 2011)</td>
<td>Proprietary</td>
<td>?</td>
</tr>
<tr>
<td>BoxedApp Packer</td>
<td>2.2 (June 16, 2009)</td>
<td>Proprietary</td>
<td>Yes</td>
</tr>
<tr>
<td>CExe</td>
<td>1.0b (July 20, 2001)</td>
<td>GPL</td>
<td>No</td>
</tr>
<tr>
<td>Enigma Protector</td>
<td>3.50 (January 12, 2012)</td>
<td>Proprietary</td>
<td>Yes</td>
</tr>
<tr>
<td>EXE Bundle</td>
<td>3.11 (January 7, 2011)</td>
<td>Proprietary</td>
<td>?</td>
</tr>
<tr>
<td>EXE Stealth</td>
<td>4.14 (June 29, 2011)</td>
<td>Proprietary</td>
<td>?</td>
</tr>
<tr>
<td>eXPressor</td>
<td>1.8.0.1 (January 14, 2010)</td>
<td>Proprietary</td>
<td>?</td>
</tr>
<tr>
<td>MPRESS</td>
<td>2.19 (January 2, 2012)</td>
<td>Freeware</td>
<td>Yes</td>
</tr>
<tr>
<td>Obsidium</td>
<td>1.4.5 (January 9, 2012)</td>
<td>Proprietary</td>
<td>Yes</td>
</tr>
<tr>
<td>PELock</td>
<td>1.0.694 (January 23, 2012)</td>
<td>Proprietary</td>
<td>No</td>
</tr>
<tr>
<td>PESpin</td>
<td>1.33 (May 3, 2011)</td>
<td>Freeware</td>
<td>Yes</td>
</tr>
<tr>
<td>RLPack Basic</td>
<td>1.21 (October 31, 2008)</td>
<td>GPL</td>
<td>No</td>
</tr>
<tr>
<td>Smart Packer Pro</td>
<td>1.7 (November 5, 2011)</td>
<td>Proprietary</td>
<td>Yes</td>
</tr>
<tr>
<td>Themida</td>
<td>2.1.9.0 (October 20, 2011)</td>
<td>Proprietary</td>
<td>?</td>
</tr>
<tr>
<td>UPX</td>
<td>3.08 (December 12, 2011)</td>
<td>GPL</td>
<td>No</td>
</tr>
<tr>
<td>VMProtect</td>
<td>2.1 (September 28, 2011)</td>
<td>Proprietary</td>
<td>Yes</td>
</tr>
<tr>
<td>XComp/XPack</td>
<td>0.98 (February 18, 2007)</td>
<td>Freeware</td>
<td>No</td>
</tr>
</tbody>
</table>
When decompression/unpacking finishes?

Heuristic-based Unpacking

- Heuristic #1: Dump as late as possible. NtTerminateProcess
- Heuristic #2: Dump when your program generates errors. NtRaiseHardError
- Heuristic #3: Dump when program forks a child process. NtCreateProcess
- Heuristic #4: Dump when program sends/recv packets.
- ...

Issues

- Weak adversarial model, too simple to evade...
- Multi-layer packing?
When decompression/unpacking finishes?

Statistics-based Unpacking

- **Observations**
  - Statistical properties of packed executable differ from unpacked executable
  - As malware executes code-to-data ratio increases

- **Complications**
  - Code and data sections are interleaved in PE executables
  - Data directories (import tables) look similar to data but are often found in code sections
  - Properties of data sections vary with packers
Overview of Unpacker

<table>
<thead>
<tr>
<th>Static Analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decompile and analyze the logical structure, flow, and data stored within the binary itself.</td>
</tr>
<tr>
<td>Able to figure out what the packer is by fingerprinting the structure of the code</td>
</tr>
<tr>
<td>It is difficult, if possible, to reveal the binary code without real execution</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dynamic analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td>Monitor the behavior of the malware binary at runtime.</td>
</tr>
<tr>
<td>Fine-grained monitor (Instruction-level)</td>
</tr>
<tr>
<td>Coarse-grained monitor (page-level)</td>
</tr>
</tbody>
</table>
Generic Automatic Unpackers

- **ACSAC 2006**: PolyUnpack
- **Worm 2007**: Renovo
- **ACSAC 2007**: OmniUnpack
- **ESORICS 2008**: Eureka
- **RAID 2008**: Justin

**Categories**

- Instruction-level
- Page-level
- System call level

**Triggers**

- Model-base trigger
- Heuristic trigger
- Statistical trigger
- Stack/environment trigger

**Performance**

- Slow
- Fast
Unpacker Case Study: The Eureka Framework

- Novel unpacking technique based on coarse grained execution tracing
- Heuristic-based and statistic-based upackaging
- Implements several techniques to handle obfuscated API references
- Multiple metrics to evaluate unpack success
- Annotated call graphs provide bird’s eye view of system interaction

http://eureka.cyber-ta.org/
The Eureka Workflow

- Packed Binary
  - Trace Malware syscalls in VM
    - Syscall trace
    - Heuristic based offline analysis
    - Statistics based Evaluator
  - Favorable execution point
- Eureka’s Unpacker
- Dis-assembly IDA-Pro
- Un-packed Binary
- Dis-assembly IDA-Pro
- Packed .ASM
- Statistics based Evaluator
- Un-pack Evaluation
- Raw unpacked Executable
  - Unknown OEP
  - No debug information
  - Unresolved library calls
  - Snapshot of data segment
  - Unreachable code
  - Loss of structures
Eureka

Coarse-grained execution tracing
- NtTerminateProcess
- NtCreateProcess

Statistical bigram analysis
- bigram.
Coarse-grained Execution Monitoring

- Generalized unpacking principle
  - Execute binary till it has sufficiently revealed itself, such as the event of program exit as a trigger (NtTerminateProcess implies that the unpacked malicious payload has been successfully decrypted).
  - Dump the process execution image for static analysis

- Monitoring execution progress
  - Eureka employs a Windows driver that hooks to SSDT (System Service Dispatch Table)
  - Callback invoked on each NTDLL system call
  - Filtering based on malware process pid
Problems

- Not all malware exit and keep an executing version resident in memory
  - Packers can make spurious event of creating new process.
  - Malware authors can simply avoid exiting the malware process.
  - The above two simple heuristics may work for a large fraction of malware today (as much as 80%), it may not be the same for future malware.
Statistical bigram analysis

- Mining statistical patterns in x86 code
  - Use simple n-gram analysis
  - Use the IDA Pro to extract regions from executable that were marked as functions.
  - Looking for the most common bigrams (opcode pairs or 2-byte opcodes) and space bigrams (byte pairs separated by 1 or more bytes)
  - Found FF 15(call), FF 75(push), E8—00 and E8—FF are prevalent in x86 code.
## Occurrence summary of bigrams

<table>
<thead>
<tr>
<th></th>
<th>calc</th>
<th>explorer</th>
<th>notepad</th>
<th>ping</th>
<th>shutdown</th>
</tr>
</thead>
<tbody>
<tr>
<td>FF 15(call)</td>
<td>246</td>
<td>3045</td>
<td>415</td>
<td>58</td>
<td>132</td>
</tr>
<tr>
<td>FF 75(push)</td>
<td>235</td>
<td>2494</td>
<td>245</td>
<td>41</td>
<td>85</td>
</tr>
<tr>
<td>E8---FF(call)</td>
<td>1583</td>
<td>2201</td>
<td>180</td>
<td>87</td>
<td>49</td>
</tr>
<tr>
<td>E8---00(call)</td>
<td>746</td>
<td>1091</td>
<td>108</td>
<td>57</td>
<td>66</td>
</tr>
</tbody>
</table>
Code-to-data ratio

Grey area stand for data
Blue area stand for code

Original `notepad.exe` memory space

Packed `notepad.exe` memory space
**Bigram Counts**

- There are consistent and significant shifts in the bigram counts.
- The simple bigram counting approach had over a 95% success rate in distinguishing between packed and unpacked malware instance.
Evaluation (ASPack)
Evaluation (MoleBox)
Evaluation (Armadillo)
## Evaluation

<table>
<thead>
<tr>
<th>Packer</th>
<th>PolyUnpack Unpacking</th>
<th>Renovo Unpacking</th>
<th>Eureka Unpacking</th>
</tr>
</thead>
<tbody>
<tr>
<td>Armadillo</td>
<td>×</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td>Aspack 2.12</td>
<td>⊗</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Asprotect 1.35</td>
<td>⊗</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td>ExeCryptor</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>ExeStealth 2</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>FSG 2.0</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>MEW 1.1</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>MoleBoxPro</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Morphine 1.2</td>
<td>✓</td>
<td>⊗</td>
<td>✓</td>
</tr>
<tr>
<td>Obsidium</td>
<td>×</td>
<td>×</td>
<td>✓</td>
</tr>
<tr>
<td>PeCompact 2</td>
<td>×</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Themida</td>
<td>×</td>
<td>⊗</td>
<td>×</td>
</tr>
<tr>
<td>UPX 3.02</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>WinUPack 3.99</td>
<td>⊗</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Yoda 3.53</td>
<td>⊗</td>
<td>⊗</td>
<td>✓</td>
</tr>
</tbody>
</table>
Systematic Approach **Unpacking**

- **Automatic Unpacking**: involves *running the malware* and capturing its memory image.
- Monitoring the execution of the malware is an *intrusive process* and is often detected using *anti-tracing* and *anti-debugging* techniques embedded in the malware.
- **Eureka** consists of minimal monitoring and capturing the process image at key events:
  - ExitProcess
  - Byte bigram monitoring: call, push instructions for instance
  - Number of seconds elapsed
  - Run the malware without monitoring and suspend its execution and perform memory inspection

- In practice, **Eureka** always manages to get a dump (memory snapshot) of the running process: *no OEP* and *no Import table*
1. Background

2. Malware Reverse Engineering
   - Unpacking
   - Disassembly
   - Decompilation
   - Program Understanding

3. Summary
Phase 2: Disassembly

The disassembler reads the PE data structure in order to:

1. Determine the different sections of the file and separate code from data and identifies resource information such as import tables
   - The disassembler relies on the PE data structure (could be corrupt)
   - The disassembler translates into code, any referenced address from known code location

2. Translate code segments into assembly language
   - The disassembler relies on the hardware instruction set documentation

3. Interpret data according to identified types
   - A data referenced by code can be of any type: integer, string, struct, etc.

Integer:
0x0040F45C dword_40F45C dd 0E06D7363h, 1, 2 dup(0) ; DATA XREF: 408C98

String:
0x0040F45C unk_40F45C db 63h ; c ; DATA XREF: sub_408C98
0x0040F45D db 73h ; s
0x0040F45E db 6Dh ; m
0x0040F45F db 0E0h ; a
IDA Pro Disassembler

- http://www.hex-rays.com/idapro/
  - It supports a variety of executable formats for different processors and operating systems. It also can be used as a debugger for Windows PE, Mac OS X, and Linux ELF executables.
  - IDA performs a large degree of automatic code analysis to a certain extent, leveraging cross-references between code sections, knowledge of parameters of API calls, limited dataflow analysis, and recognition of standard libraries.
    - Hashes of known statically linked libraries are compared to hashes of identified subroutines in the code
    - Provides scripting languages to interact with the system to improve the analysis.
  - Support plug-ins: The IDA decompiler is the most impressive plug-in.
PE Execution

1. Read the Portable Executable (PE) file data structure and maps the file into memory
2. Load import modules
3. Start execution at entry point
4. Runtime unpacking
5. Jump to OEP
Fixing the Disassembled Code

- Unpacked & disassembled code does not have an OEP, as shown in Eureka framework.
- Import tables are rebuilt dynamically and there are no static references to dynamically loaded libraries.
- Header information is not reliable.
- Data is not typed.
Parsing the PE executable format
Challenges in Binary Code Disassembly

- Disassembly is not an exact science: On CISC platforms with variable-width instructions, or in the presence of self-modifying code, it is possible for a single program to have two or more reasonable disassemblies. Determining which instructions would actually be encountered during a run of the program reduces to the proven-unsolvable halting problem.

- Bad disassembly because of variable length instructions
- Jumps into middle of instructions
- No reachability analysis: Unreachable code can hide data.
Examples of Disassembly problems (The Storm Worm)

Data hidden as code:

```
ArcadeWorld.exe:0042BDB0  mov     eax, 0
ArcadeWorld.exe:0042BDB5  test    eax, eax
ArcadeWorld.exe:0042BDB7  jnz     short loc_42BDD8; unreachable code

ArcadeWorld.exe:0042BDD8  lea     edx, [ebp+401C42h]
ArcadeWorld.exe:0042BDDE  lea     eax, [ebp+40B220h]
ArcadeWorld.exe:0042BDE4  push    eax
ArcadeWorld.exe:0042BDE5  push    8A20h
ArcadeWorld.exe:0042BDEA  push    edx
ArcadeWorld.exe:0042BDEB  call    near ptr unk_42BF7D
ArcadeWorld.exe:0042BDF0  pusha
ArcadeWorld.exe:0042BDF1  mov     edi, [ebp+40C067h]
ArcadeWorld.exe:0042BDF7  add     edi, [ebp+40C03Fh]
ArcadeWorld.exe:0042BDFD  lea     esi, [ebp+40BFBFh]
```
User-level malware programs require system calls to perform malicious actions.

- Use Win32 API to access user level libraries.
- Obfuscations impede malware analysis using IDA Pro or OllyDbg.
  - Packers use non-standard linking and loading of dlls.
  - Obfuscated API resolution.
Standard API Resolution

Imports in IAT identified by IDA by looking at Import Table
Handling Thunks

- Identify subroutines with a JMP instruction only
- Treat any calls to these subs as an API call
Leveraging Standard API Address Loading

Function Name : ADSICloseDSObject
Address : 0x76e30826
Relative Address : 0x00020826
Ordinal : 142 (0x8e)
Filename : adsldpc.dll
Full Path : c:\WINDOWS\system32\adsldpc.dll
Type : Exported Function

Function Name : ADSICloseSearchHandle
Address : 0x76e3050a
Relative Address : 0x0002050a
Ordinal : 143 (0x8f)
Filename : adsldpc.dll
Full Path : c:\WINDOWS\system32\adsldpc.dll
Type : Exported Function

Function Name : ADSICreateDSObject
Address : 0x76e30447
Relative Address : 0x00020447
Ordinal : 144 (0x90)
Filename : adsldpc.dll
Full Path : c:\WINDOWS\system32\adsldpc.dll
Type : Exported Function
Using Dataflow Analysis

Identify register based indirect calls

GetEnvironmentStringW
Handling Dynamic Pointer Updates

Identify register based indirect calls

A def to dword_41e308 is found
Look for probable call to GetProcAddress earlier

Call to GetProcAddress

dword_41e304 has no static value to look up API
Rebuilding the unpacked executable

- From a damaged dumped image of a running malware to a PE executable:
  - Knowing all APIs allows us to identify the OEP.
  - Semantic approach: ExitProcess, CreateMutex, GetCommandLine, GetModulaHandle, etc are close to OEP. There are about 20 APIs that are often called at the beginning of the execution of the code.
  - Structural approach: find sources of call graphs in the binary
  - Rebuilding in import table with all references to identified APIs

- The disassembly of the reconstructed PE is often of better quality than the disassembly of the dumped process image
  - The new PE code bypasses the unpacking routine embedded in the packed code
  - The new PE contains the original code
Outline

1. Background

2. Malware Reverse Engineering
   - Unpacking
   - Disassembly
   - Decompilation
   - Program Understanding

3. Summary
Phase 3: Decompilation

- Identifies local variables
- Identifies arguments: registers, stack, or any combination
- Identifies global variables
- Identify calling conventions
- Identifies common idioms and compiler features
- Eliminates the use of registers as intermediate variables
- Identifies control structures
Decomposition Depends on Previous Analysis Phases

```
LPVOID __stdcall sub_4032B4(void *a1)
{
    char Source; // [sp+0h] [bp-14h]+1
    DWORD nSize; // [sp+10h] [bp-4h]+1

    nSize = 16;
    GetComputerNameA(&Source, &nSize);
    return sub_401943(a1, &Source);
}
```
Analysis Phases

**Ideally**
- Source Code
  - Compiler
  - Executable code
  - Disassembly & Analysis
  - Assembly code
  - Decompilation
  - Legitimate C/C++ that a compiler would generate

**Reality**
- Malware
  - Unpacking
  - Non-executable code
  - Disassembly & Analysis
  - Obfuscated assembly
  - Decompilation
  - A mess
  - Undo Obfuscation
  - Assembly code
  - Decompilation
  - Legitimate C/C++
Example of Binary Rewrite

```c
bool __usercall is_private_subnet (unsigned __int16 a1) {
  return a1 == 43200 || a1 == 10 || (a1 & 0xF0FF) == 4268;
}
```
1. Background

2. Malware Reverse Engineering
   - Unpacking
   - Disassembly
   - Decompilation
   - Program Understanding

3. Summary
Phase 4: Program Understanding

- Need to identify higher-level concepts from the deobfuscated code
- Need to interpret the code into a higher-level malware objective
- Need to identify particular features: crypto:
  - Functions that use crypto-related opcode, loops, etc
  - Known constants in crypto algorithms
Finding Known and Unknown Crypto
Outline

1 Background

2 Malware Reverse Engineering
   - Unpacking
   - Disassembly
   - Decompilation
   - Program Understanding

3 Summary
Malware analysis: Deobfuscation

Malware author designs obfuscation to

- Packing: to reduce the size of binaries and to create polymorphic malware samples
- Besides packing, the more advanced obfuscation techniques are designed to slow down reverse engineering efforts and to prevent:
  - the identification of API calls: identify the basic building blocks of the malware
  - the control-flow reconstruction of the malware: follow and reconstruct the logic flow
  - static analysis: determine the full functionality, triggers, hidden logic, time bombs, etc.
Why Code Obfuscation is not Easy

- Malware authors can design binary code that is extremely difficult to analyze. Using advanced programming languages knowledge, it is possible to create such code.
- Malware code should be able to run in a reliable manner. Obfuscation should not compromise this important requirement and should maintain the reliability of the initial code. This requires a proof or guarantee of some sort.
Stuxnet: Keeping it “relatively” simple

- Stuxnet does not use advanced binary obfuscation techniques.
- The analysis of the code is challenging nevertheless

Stuxnet Code Characteristics:

- Use of C++
- Use of C++ exception handling
- Use of C++ classes
- Use of simple data encoding (encryption)
- Use of C structures for all data passed to the main subroutines:
  - Over 40 user-defined structures
  - Not recognized by disassemblers and decompilers
It is always desirable to recover from the malware a description that is as close as possible to the original code produced by the authors.

It is often possible to do that in practice.

It is often the only way to really determine the full capability of the malware.

Malware analysis: deobfuscation + reverse engineering

Unpacking, Disassembling, Decompilation
Further Reading

- Renovo: A hidden code extractor for packed executables. In WORM 2007