Background Malware Reverse Engineering Summary
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 Reverse Engineering
   - Unpacking
   - Disassembly
   - Decompilation
   - Program Understanding
3 Summary
## Background

### 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

Credit: Some 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

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

### Typical features of a malware binary:
- 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

### What does the malware do

| 01001010100101010 | 10101010011010101 |
| 01001010100101010 | 10101010011010101 |
| 01001010100101010 | 10101010011010101 |
| 01001010100101010 | 10101010011010101 |
| 01001010100101010 | 10101010011010101 |
| 01001010100101010 | 10101010011010101 |
| 01001010100101010 | 10101010011010101 |
| 01001010100101010 | 10101010011010101 |

`.exe`
Dynamic vs Static Malware Analysis

**Dynamic Analysis**
- Techniques that profile actions of binary at runtime
- More popular
  - CW Sandbox, TT Analyze, 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

Malware Reverse Engineering

**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.
### 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.
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


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?

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

Available Packer

<table>
<thead>
<tr>
<th>Name</th>
<th>Latest stable</th>
<th>Software license</th>
<th>std-64 support</th>
</tr>
</thead>
<tbody>
<tr>
<td>JoelSummer iPA</td>
<td>2.0 (March 26, 2012)</td>
<td>Proprietary</td>
<td>Yes</td>
</tr>
<tr>
<td>Armadillo Packer</td>
<td>8.60 (July 8, 2011)</td>
<td>Proprietary</td>
<td>Yes</td>
</tr>
<tr>
<td>ASPack Outreach</td>
<td>2.28 (May 18, 2011)</td>
<td>Proprietary</td>
<td>?</td>
</tr>
<tr>
<td>ASPack/ASP/Outlook</td>
<td>1.04 (September 1, 2011)</td>
<td>Proprietary</td>
<td>?</td>
</tr>
<tr>
<td>Boomtape Packer</td>
<td>2.2 (June 16, 2008)</td>
<td>Proprietary</td>
<td>Yes</td>
</tr>
<tr>
<td>CDex</td>
<td>1.09 (July 20, 2001)</td>
<td>GPL</td>
<td>No</td>
</tr>
<tr>
<td>Brijinn Protector</td>
<td>3.06 (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>iXPressor</td>
<td>1.0.0.1 (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>Obfuscator</td>
<td>1.4 (January 9, 2012)</td>
<td>Freeware</td>
<td>Yes</td>
</tr>
<tr>
<td>PELock</td>
<td>10.949 (January 23, 2012)</td>
<td>Proprietary</td>
<td>No</td>
</tr>
<tr>
<td>PESpiner</td>
<td>1.32 (May 3, 2011)</td>
<td>Freeware</td>
<td>Yes</td>
</tr>
<tr>
<td>PC Pack Basic</td>
<td>1.21 (October 31, 2009)</td>
<td>GPL</td>
<td>No</td>
</tr>
<tr>
<td>Bent Packer Pro</td>
<td>1.7 (November 5, 2011)</td>
<td>Proprietary</td>
<td>Yes</td>
</tr>
<tr>
<td>Threnode</td>
<td>2.1 (October 20, 2011)</td>
<td>Proprietary</td>
<td>?</td>
</tr>
<tr>
<td>UPX</td>
<td>3.68 (December 12, 2011)</td>
<td>GPL</td>
<td>No</td>
</tr>
<tr>
<td>VIPProtect</td>
<td>2.1 (September 26, 2011)</td>
<td>Proprietary</td>
<td>Yes</td>
</tr>
<tr>
<td>XComp/XPack</td>
<td>0.99 (February 18, 2007)</td>
<td>Freeware</td>
<td>Yes</td>
</tr>
</tbody>
</table>
### Overview of Unpacker

#### Static Analysis
- Decompile and analyze the logical structure, flow, and data stored within the binary itself.
- Able to figure out what the packer is by fingerprinting the structure of the code
- It is difficult, if possible, to reveal the binary code without real execution

#### Dynamic Analysis
- Monitor the behavior of the malware binary at runtime.
  - Fine-grained monitor (Instruction-level)
  - Coarse-grained monitor (Page-level)

### Generic Automatic Unpackers

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>PolyUnpack</td>
<td>RenoUnpack</td>
<td>OmniUnpack</td>
<td>Eureka</td>
<td>Justin</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instruction-level</th>
<th>Page-level</th>
<th>System call level</th>
<th>Page-level</th>
<th>Stack/environment trigger</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model-based trigger</td>
<td>Heuristic trigger</td>
<td>Heuristic and Statistical trigger</td>
<td>Fast</td>
<td>Fast</td>
</tr>
</tbody>
</table>

### Unpacker Case Study: The Eureka Framework

- Novel unpacking technique based on coarse grained execution tracing
- Heuristic-based and statistic-based unpacking
- 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/
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

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.

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.
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.
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.
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.
  2. The disassembler relies on the PE data structure (could be corrupt).
  3. The disassembler translates into code, any referenced address from known code location.
  4. Translate code segments into assembly language.
  5. The disassembler relies on the hardware instruction set documentation.
  6. Interpret data according to identified types.

- A data referenced by code can be of any type: integer, string, struct, etc.

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

- Read the Portable Executable (PE) file data structure and maps the file into memory.
- Load import modules.
- Start execution at entry point.
- Runtime unpacking.
- 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

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.
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

**Handling Thunks**

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

**Leveraging Standard API Address Loading**

- Leveraging standard API address loading allows for more accurate malware analysis.

**Imports in IAT identified by IDA by looking at Import Table**

- Imports in IAT are identified by IDA by looking at the Import Table.

**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

Handling Dynamic Pointer Updates

Identify register based indirect calls

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, GetModuleHandle, 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

Analysis Phases

- Ideally: Source Code
- Reality: Malware
- Compiler
- Executable code
- Disassembly & Analysis
- Non-executable code
- Disassembly & Analysis
- Assembly code
- Decomposition
- Legitimate C/C++ that a compiler would generate

Example of Binary Rewrite

```c
int __usercall OBFUSCATED_VERSION_OF_is_private_subnet(unsigned __int16 a1) {
    int result; // eax@2
    if (a1 == 43200) result = 1;
    else result = off_9BAAA5();
    return result;
}
```
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

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

Summary

- 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
<table>
<thead>
<tr>
<th>Further Reading</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Renovo: A hidden code extractor for packed executables. In WORM 2007</td>
</tr>
</tbody>
</table>