Reverse Engineering Malware

Hassen Saidi
Computer Science Laboratory
SRI International
Hassen.Saidi@sri.com
The Growth of a Network

THE ARPA NETWORK
DEC 1969

4 NODES
Conceptual Sketch of Original Internet
The Growth of a Threat

Mass email campaign: Love letter, Melissa
Multiple vectors of infection, attacks against AV software,
Combined infection vectors, dangerous payloads: Code Red, Nimda

Email virus + social engineering: Xmas Exec
Large scale pandemics: Morris worm
Infected 10% of the Internet.

Self replicating
Program: Creeper

Sophisticated engineering: Conficker
Use of Crypto.
Social Networks/cell phone worms.
Stuxnet,...
Malware incidents are rising dramatically:
- increase of infection vectors
- increase in the complexity of botnet structures

From Biology: Connected World Gives Viruses The Edge
“as human activity makes the world more connected, natural selection will favor more virulent and dangerous parasites."
COMPUTER VIRUS SPREADS TO HUMANS!
Motivation

• Malware landscape is diverse and constant evolving
  – Large botnets
  – Diverse propagation vectors, exploits, C&C
  –Capabilities – backdoor, keylogging, rootkits,
  – Logic bombs, time-bombs
  – Diverse targets: desktops, mobile platforms, SCADA systems (Stuxnet)

• 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.
Outline

• Review of the workflow of binary program analysis
• Review of the challenges in binary program analysis:
  – Obfuscation Techniques
• Techniques for reverse engineering stripped binaries:
  – Systematic deobfuscation
• Examples of obfuscation: Conficker, Hydrac (Google attack), Stuxnet, ...
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 Binary Analysis

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
• 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
• …

Challenges:
• lack of automation
• time-critical analysis
• labor intensive
• requires a human in the loop
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
My Personal Philosophy

- Push the limits of static analysis as much as possible.
- Rebuild the binary in its original form prior to obfuscation.
- Recover a higher level description of the malware binary that makes deriving the purpose of the malware attainable: I want to stare at C code as opposed to staring at assembly code.
Malware Revere Engineering System Goals

• Desiderata for a Static Analysis Framework
  – 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 for instance)
Reverse Engineering Phases

- **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.

- **Disassembly phase:**
  - Identification of code and data segments
  - Relies on the unpacker to capture all code and data segments. Our unpacking approach guarantees that.

- **Decompilation phase:**
  - 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. Our API resolution guarantees that.

- **Program understanding phase:**
  - Relies on the decompiler to produce readable C code, by recognizing the compiler, calling conventions, stack frames manipulation, functions prolog and epilog, user-defined data structures. Our code rewrite and analysis guarantees that.
Phase 1: Malware Unpacking
Example of Packed Code
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
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**
  - Unpacked Binary
- **Disassembly IDA-Pro**
  - Packed Binary
  - Un-Packed ASM
  - Un-Packed Evaluator
- **Statistics based Evaluator**
  - Raw unpacked Executable
    - Unknown OEP
    - No debug information
    - Unresolved library calls
    - Snapshot of data segment
    - Unreachable code
    - Loss of structures

Unpack Evaluation
Coarse-grained Execution Monitoring

• Generalized unpacking principle
  – Execute binary till it has sufficiently revealed itself
  – 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
Heuristic-based Unpacking

• How do you determine when to dump?
  – 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

• Issues
  – Weak adversarial model, too simple to evade...
  – Doesn’t work well for package non-malware programs
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
Statistics-based Unpacking (2)

• Our Approach
  – Model statistical properties of unpacked code

• Estimating unpacked code
  – N-gram analysis to look for frequent instructions
  – We use bi-grams (2-grams) because x-86 opcodes are 1 or 2 bytes
  – Extract subroutine code from 9 benign executables
  – FF 15 (call), FF 75 (push), E8 _ _ _ ff (call), E8 _ _ _ 00 (call)
Evaluation (ASPack)
Evaluation (MoleBox)
Evaluation (Armadillo)
Systematic Approach to Code Deobfuscation: 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.

• Our multi-strategy approach 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, we always manage 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
     - 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
     1. 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
      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/](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

1. Start execution at entry point
2. Runtime unpacking
3. Jump to OEP
Phase 3: Fixing the Disassembled Code

• Unpacked & disassembled code does not have an OEP.
• 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
API Resolution

• 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

```assembly
loc_401882:
  inc   eax
  cmp   byte ptr [ecx+eax], 0
  jnz   short loc_401882

loc_401831:
  cmp   [ebp+var_8], eax
  jb    loc_401831

loc_401894:
  call  sub_40C634
  end   [ebp+arg_4], 0
  call  sub_40C598
  xor   ebx, ebx
  jmp   short loc_401894

loc_401894:
  mov   eax, [ebp+arg_4]
  mov   ecx, eax
```

```assembly
Model: 004473CC
data: 7C836634
Model: 004473D0
data: 7C891072
Model: 004473D4
data: 7C8286EE
Model: 004473E0
data: 7C80674A
Model: 004473E4
data: 7C80F310
Model: 004473E8
data: 7C80F979
Model: 004473EC
data: 7C813093
Model: 004473F0
data: 7C81D777
Model: 004473F4
data: 7C8099DA
Model: 004473F8
data: 7C8099F9
Model: 004473FC
data: 7C809E1B
Model: 00447400
data: 7C80A124
```

```
Model: 004473E4
data: dword_4473E4
```

```
Model: 004C62E
data: 2 dup(90h)
Model: 004C630
data: 0
Model: 004C634
data: ; SUBROUTINE === Attributes: thunk
Model: 004C634
data: sub_40C634 proc near
Model: 004C634
data: jmp ds:dword_4473E4
```

```
Model: 004C634
data: sub_40C634 endp
```
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
Leveraging Standard API Address Loading is not enough

There are many indirect ways to load and call a Windows API:
• access to list of loaded DLLs
• access to a loaded DLL and use of GetModuleHandle() + offset
• …
Consequence of Failure to Identify APIs

... push offset unk_40A2DC ; arg 1
   xor ebx, ebx
.text:004011AE call dword ptr unk_40A0E4 .data:0040A0E4 00000000
.text:004011B4 mov edi, eax
.text:004011B6 cmp edi, ebx
.text:004011B8 jz short loc_401211
.text:004011BA push esi
.text:004011BB mov esi, dword ptr unk_40A0E8
.text:004011C1 push offset unk_40A2C4 ; arg 2
.text:004011C6 push edi ; arg 1
.text:004011C7 call esi ; unk_40A0E8 .data:0040A0E8 00000000
.text:004011C9 push offset unk_40A2AC
.text:004011CE push edi
.text:004011CF mov dword_433480, eax

... lea eax, [ebp+var_4]
.text:00401135 push eax
.text:00401136 push ebx
.text:00401137 push 0
.text:00401139 mov [ebp+var_4], esi
.text:0040113C call dword_433480
.text:00401142 test eax, eax

Name of a library
Load library call (LoadLibrary)
Name of the library function
Name of the library
API call to get the address
Of the loaded library function (GetProcAddress)
library function call
Failure to Perform Control Flow Analysis

- CreateThread

```
.text:009A3A4C    push  eax
.text:009A3A4D    xor  eax, eax
.text:009A3A4F    push  eax
.text:009A3A50    push  eax
.text:009A3A51    push  offset dword_9A3939
.text:009A3A56    push  eax
.text:009A3A57    push  eax
.text:009A3A58    call  [ebx]
.data:009A3939    xxxxxxx
```

- Location of the start address of a thread
- Call to CreateThread

- Starting Services
- Thread synchronization
- Critical sections
- Callback functions
Advanced API Resolution

• There are many ways in which a library or API can be invoked.
• There are many ways an API call can be obfuscated
• But there is one invariant associated to each API and library: its signature
  – i.e; number of arguments, type of arguments, and type of return value if any.
Advanced API Resolution: Type Inference for binary program analysis

• Use type inference as a single solution to solve three fundamental problems:
  – Identifying API and function calls (call and jump targets)
  – Building a precise CFG
  – Recovering user-defined types for proper decompilation

• For Windows Executable files:
  – Integers: object handles, addresses, IP address, ports, etc
  – Strings: file names, service names, etc
  – Structures: sockaddr

```c
struct sockaddr_in {
    short sin_family;
    u_short sin_port;
    struct in_addr sin_addr;
    char sin_zero[8];
};
```
Type propagation and matching

- Type propagation using dataflow analysis
- Propagation of return values and arguments of functions

There is only one API that has 7 arguments such that the seventh and third and first one can be pointers and all others are not.

HANDLE WINAPI CreateFile(
    __in    LPCTSTR lpFileName,
    __in    DWORD dwDesiredAccess,
    __in    DWORD dwShareMode,
    __in    LPSECURITY_ATTRIBUTES lpSecurityAttributes,
    __in    DWORD dwCreationDisposition,
    __in    DWORD dwFlagsAndAttributes,
    __in    HANDLE hTemplateFile
);
Advantages of type Inference Analysis

• Programmers data structures and types are going to be based on known data structures and types provided by the libraries

• Identifying API calls and type information help capture better the semantics of the program execution

• Not restricted to Windows but require knowledge of the libraries and their documentation

• Can deal with some of the widely used obfuscation techniques
  – Import table obfuscation
  – Code rewrite: code rewrite preserves the types!
Phase 3: 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
.text:00401010 proc near ; CODE XREF: .text:0040550C1
.text:00401010
.text:00401010 var_B4 = dword ptr -0B4h
.text:00401010 var_B0 = dword ptr -0B0h
.text:00401010 var_AC = dword ptr -0ACH
.text:00401010 var_A8 = dword ptr -0A8h
.text:00401010 var_A4 = dword ptr -0A4h
.text:00401010 var_A0 = dword ptr -0A0h
.text:00401010 var_90 = dword ptr -90h
.text:00401010 var_4 = dword ptr -4

.text:00401010 push ebp
.text:00401011 mov ebp, esp
.text:00401013 and esp, 0FFFFFF8h
.text:00401016 sub esp, 0B4h
.text:0040101C mov eax, dword_413034
.text:00401021 xor eax, esp
.text:00401023 mov [esp+0B4h+var_4], eax
.text:0040102A push ebx
.text:0040102B push esi
.text:0040102C push edi
.text:0040102D lea eax, [esp+0C0h+var_B4]
.text:00401031 push eax ; _DWORD
.text:00401032 mov ebx, 32h
.text:00401037 call ds:dword_40F0BC
.text:0040103D push eax ; _DWORD
.text:0040103E call ds:dword_40F1BC
.text:00401044 mov esi, eax
.text:00401046 xor eax, eax
.text:00401048 push offset sub_401000 ; _DWORD
.text:0040104D mov [esp+0C4h+var_B0], offset aCbeutsvc ; "CBeutSvc"
.text:00401055 mov [esp+0C4h+var_AC], offset sub_404110
.text:0040105D mov [esp+0C4h+var_A8], eax
.text:00401061 mov [esp+0C4h+var_A4], eax
.text:00401010 ; int __stdcall WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nShowCmd)
.text:00401010 _WinMain@16 proc near ; CODE XREF: start+1724p
.text:00401010 .pNumArgs = dword ptr -0B4h
.text:00401010 .ServiceStartTable = SERVICE_TABLE_ENTRYA ptr -0B0h
.text:00401010 var_A8 = dword ptr -0B8h
.text:00401010 var_A4 = dword ptr -0B4h
.text:00401010 VersionInformation = _OSVERSIONINFOA ptr -0A8h
.text:00401010 var_4 = dword ptr -4
.text:00401010 hInstance = dword ptr 8
.text:00401010 hPrevInstance = dword ptr 0Ch
.text:00401010 lpCmdLine = dword ptr 10h
.text:00401010 nShowCmd = dword ptr 14h
.text:00401010 push ebp
.text:00401011 mov ebp, esp
.text:00401013 and esp, 0FFFFFFF8h
.text:00401016 sub esp, 0B4h
.text:0040101C mov eax, dword_413034
.text:00401021 xor eax, esp
.text:00401023 mov [esp+0B4h+var_4].eax
.text:0040102A push ebx
.text:0040102B push esi
.text:0040102C push edi
.text:0040102D lea eax, [esp+0C0h+pNumArgs]
.text:00401031 push eax ; pNumArgs
.text:00401032 mov ebx, 32h
.text:00401037 call GetCommandLineW
.text:0040103D push eax ; lpCmdLine
.text:0040103E call CommandLineToArgvW
.text:00401044 mov esi, eax
.text:00401046 xor eax, eax
.text:00401048 push offset TopLevelExceptionFilter ; lpTopLevelExceptionFilter
.text:0040104D mov [esp+0E4h*ServiceStartTable.lpServiceName], offset ServiceName ; "CBruntSvc"
.text:00401055 mov [esp+0E4h*ServiceStartTable.lpServiceProc], offset sub_404110
Phase 4: 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

```assembly
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);
}
```
Malware Obfuscation Effect on Decompilation

• While packing is the most used obfuscation technique, it is often combined with other advanced forms of obfuscation that make decompilation often impossible:
  • Call obfuscation in general and API obfuscation in particular
  • Binary Rewrite to create semantically equivalent code with vastly different structure
  • Chuncking or “code spaghettisation”
  • ...
**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
            - Assembly code
              - Undo Obfuscation
                - A mess
          - Decompilation
            - Legitimate C/C++
Example of Binary Rewrite

bool __usercall is_private_subnet (unsigned __int16 a1) {
    return a1 == 43200 || a1 == 10 || (a1 & 0xF0FF) == 4268;
}
Systematic Approach to Code Deobfuscation:
Binary Rewriting

• **Dechunking:** The control flow of Conficker's P2P module has been significantly obfuscated to hinder its disassembly and decompilation. Specifically, the contents of code blocks from each subroutine have been extracted and relocated throughout different portions of the executable. These different blocks (or chunks) are then referenced through unconditional and conditional jump instructions. In effect, the logical control flow of the P2P module has been obscured (spaghetti-code) to a degree that the module cannot be decompiled into coherent C-like code, which typically drives more in-depth and accurate code interpretation. **Move all blocks to a contiguous memory block.**

• **Normalize x86 instructions:** *push followed by a pop is a mov*

• **Normalize calling convention:** *cdecl,fastcall, stdcall,* instead of user-defined.
Conficker and Hydrac Dechunking

- Identify all chunks in a function and rewrite the function
- Applied to all Conficker C P2P Protocol subroutines
- Unlike the Conficker P2P logic, Hydraq did not exhibit the same level of obfuscation. It did, however, share some obfuscation features with Conficker. The functions of the Hydraq binary have been subjected to chunking, which renders decompilation difficult. We applied our transformations to automatically generate the C-like code for each subroutine and build a complete CFG of the binary. The IDA disassembler identified 185 subroutines in the binary prior to our analysis. After running the dechunking transformation, only 141 subroutine remained and were decompiled.
Purpose of code obfuscation

• While packing is often used to reduce the size of binaries and to create polymorphic malware samples, 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.
  – timely reverse engineering and mitigation of the threat
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 authors do not feel the need to always obfuscated their code. Can easily defeat signature-based detection. Overwhelm analysts and tools with large numbers of samples.
- 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.
- Malware deobfuscation is therefore a more attainable than you might think. Systematic obfuscation informs systematic deobfuscation.
Our Approach

- Because obfuscation is introduced in a rather systematic way, there is a hope that it can be dealt with in an automated way.

- Systematically identifying an obfuscation step and undoing its effect.

- Focus on generic approaches as opposed to packer/obfuscator specifics
Example: Static Analysis of Conficker

- Conficker appeared on November 20th, 2008
- Infected millions of machines worldwide
- Millions of machines still infected despite an extensive news coverage about the threat
- Four versions have been released: A, B, B++, and C
- It is a sophisticated piece of malicious code created by professionals who have extensive knowledge about networking, cryptography, system and network programming, and security
- Managed to defeat the security community in stopping its progression by using strong crypto, code obfuscation, aggressive propagation strategies, and constantly monitoring the security community actions
- Dynamic analysis provided a limited understanding of the threat:
  - Identification of what appears to be a P2P protocol
  - Identification of ports opened by the malware
- Deobfuscation and static analysis were the only techniques that were able to uncover the full capability of the malware.

Example: Static Analysis of Conficker

Static Analysis of Conficker Code:

- Domain generation algorithm: *provided a list of daily domains to be blocked*
- Quarter of the Internet scanned: *Understand what part of the Internet was targeted for scanning and what infections were due to USB ports and mobile devices*
- List of disabled security products: *detection*
- Ukrainian keyboard avoidance: *Geo-location database poisoning*
- Use of MD6 and related crypto algorithms: *Attribution*
- DNS APIs patching to disable list of websites (including SRI!): *detection*
- Distribute a number of modified versions of the binary
- TCP and UDP ports based on the IP address of the infected machine: *detection*
Deobfuscation of the Conficker C P2P protocol

- Heavily obfuscated protocol code
- 88 APIs obfuscated
- Use of chunking lead to poor decompilation
- Benefits of the deobfuscation
  - P2P Protocol description: protocol understanding and P2P structure
  - Peer selection algorithm: proved the peer poisoning approach useless
  - Possibility to hot patch code without DGA updates: proved C&C domains obsolete

The P2P protocol was not just a mechanism for distributing PE executable files but also digitally signed sets of x86 instructions that are executed in a separate thread and take as argument the IP address of the sender. This would provide a hot patch mechanism for all data manipulated by Conficker: list of peers, encryption/decryption keys, the Conficker code itself, etc.
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
Phase 5: 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 indentify particular features: crypto:
  – Functions that use crypto-related opcode, loops, etc
  – Known constants in crypto algorithms
Finding Known and Unknown Crypto
Conclusions

• 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
• The benefits are important when it comes to high-profile targets
• Easily integrated in common analysis tools: disassembler (IDA), Decompilers.
Daily Malware Capture and Analysis

• http://mtc.sri.com/