Automatic Reverse Engineering of Malware Emulators

Monirul Sharif, Andrea Lanzi, Jonathon Giffin, Wenke Lee

School of Computer Science
College of Computing
Georgia Institute of Technology, USA
{msharif, andrew, jon, wenke}@cc.gatech.edu

(to appear in IEEE SYMPOSIUM on SECURITY and PRIVACY, May 2009)
Malware Obfuscation

- **Anti-analysis techniques continuously evolve**
  - Malware authors adopt new techniques to defeat analysis

- **Code obfuscation**
  - Code encryption, polymorphism – unpacks entire code once
  - Multi-layered packing – several levels of unpacking
  - Page-by-page – unpacks code page by page on demand
  - Trend is moving towards finer grained code obfuscation

- **Recent move towards emulator-based obfuscation**
  - An instruction-level obfuscation approach
  - Several commercial packers support emulator based obfuscation, including Code Virtualizer and VMProtect
  - Emulation techniques maturing, widespread adoption is possible
Emulator-Based Obfuscation

- $P_{x86}$: Original Malware Program (written in x86)
- $EM_{x86}$: Emulator
- $PL$: Bytecode Program (written in language $L$)
- ISA for $L$: Instruction Set Architecture for language $L$

Translate from x86 to language $L$
Impacts on Existing Malware Analysis

- **Unknown Language** $L$
  - $L$ can be randomly generated

- **Pure Static Analysis (whitebox)**
  - Completely thwarted
  - Only emulator code is analyzable
  - $P_L$ is considered as data by analyzer

- **Greybox methods**
  - Includes instruction level analyzers, information-flow, dynamic tainting, multi-path exploration etc.
  - Analysis is inaccurate
  - For example, paths may explored in the emulator, but not the malware
Motivations

- **Manual reverse-engineering methods cannot scale**
  - Manual reverse-engineering takes time
  - Each malware instance can have new bytecode language and emulator, making reverse-engineered information obsolete

- **Need automated techniques to reverse emulator**
  - Should not require any knowledge about bytecode
  - Should be generic and work for a large class of emulators

- **Is automated reverse engineering possible?**
  - Theoretically, it is an undecidable problem
  - However, from intuition, the emulator’s fetch, decode and execution behavior can be identified at run-time
**Approaches of Emulation**

**Decode-Dispatch Emulation**

**Byte code**

**Decode-dispatch** emulation utilizes a central loop to fetch, decode and execute the bytecode instructions.

The **VPC** (Virtual Program Counter) is maintained to point to the next bytecode instruction to fetch from.

**x86 code**

while (vpc != end) {
    opcode = P[vpc];
    execfunc = TBL[opcode];
    execfunc();
}

execADD() {
    opr1 = P[vpc+1];
    opr2 = P[vpc+2];
    opr3 = P[vpc+3];
    R[opr1] = R[opr2] + R[opr3];
    vpc += 4;
}

eexecCJMP() {
    T1 = vpc + 5;
    T2 = *(DWORD*) &P[vpc+1];
    F = R[CFLAG]; // 0 or 1
    vpc = T1 + (T2 - T1) * F;
}

eexecJUMP() {
    T = *(DWORD*) &P[vpc+1];
    vpc = *addr;
}

Jump table usage depicting switch like constructs

Address computation without conditional branch instruction
Challenges

- No knowledge of bytecode program
  - The location of the bytecode program in the obfuscated programs memory is not known

- No knowledge of emulators code
  - The code that corresponds to decode, dispatch and execute phases of the emulator is not known

- Intentional variations
  - Context can be maintained in many different ways
  - An attacker may complicate VPC identification by maintaining it in different correlated variables
  - Bytecode program may be stored in non-contiguous memory
Approach Overview

- **Abstract Variable Binding**
  - Identify pointer variables within raw memory of emulator using access patterns of memory reads and writes
  - It is a combination of forward and backward data-flow analysis

- **Identifying Candidate VPCs**
  - Cluster memory reads according to their bound variables
  - Each cluster provides a candidate VPC

- **Identify Emulator Phases**
  - Identify decode-dispatch loop and emulator phases

- **Extract Bytecode Syntax and Semantics**
  - Identify fetched instruction syntax – opcode and operands
  - Identify execute routine related to instruction – this is the semantics of the bytecode instruction
  - This information is subsequently used to generate CFGs
Automatically Reverse Engineering Malware Emulators

Sharif et al.

Abstract

Memory reads bound to abstract variables

Clustered memory reads. Each cluster is a bytecode candidate

Emulator phase identification

Bytecode instruction syntax, semantics and bytecode CFG

Bytecode syntax and semantics extraction

Fetch, decode and execute routines

Approach Workflow
Abstract Variable Binding

- **Characterizing pointer variables in code**
  - The VPC variable maintains the address of bytecode and is the key to our analysis
    
    \[
    \text{instruction} = \text{bytecode[vpc]} 
    \]
  - The challenge is that in the binary level, the notion of variables are lost
  - We associate reads with their pointer variables as abstract variable binding

- **Forward and backward binding**
  - Each memory location is a possible abstract variable
  - Abstract variables are bound to registers or other variables possibly holding their values
  - Forward and backward binding propagates binding information in a conservative manner (Details are in the paper)
Identifying Candidate VPCs

- **Abstract variables are correlated**
  - If an abstract variable influences another, they are called dependent abstract variables

- **Cluster generation**
  - We cluster memory reads that have at least one common dependent abstract variable
We identify fundamental properties of decode-dispatch emulators

- The central decode-dispatch loop, containing opcode fetch using VPC
- Dispatch routine that uses opcode to dispatch execute routines
- Execute routines that execute the semantics of the instruction

We use taint analysis and state machines

- We use multi-label tainting to taint each clustered read address with a separate label \(<vpc, id>\)
- Taint is propagated through execution
- If a tainted data influences control-flow, we mark as dispatch behavior. Targets of the jumps are marked as execute routines.
- At least two iterations of a loop containing the decode and dispatch like behavior confirms an emulation loop
Each iteration of decode-dispatch loop is analyzed for syntax and semantics of fetched bytecode

**Syntax identification:**
- *Opcode* is identified from dispatch behavior (jump is made based on opcode)
- Any reads from bytecode (constant offset from opcode) within execute routine indicate *operand*. We can identify at byte-level

**Semantics identification:**
- Each execute routine is considered to correspond to a specific bytecode instruction
- The x86 code of an execute routine is considered the semantics
- Control-flow semantics – updates made to the VPC are control-flow semantics
Experimental Evaluation

Bytecode syntax and semantics identification evaluation (synthetic program)

Original x86 CFG
Code Virtualizer Bytecode CFG
VMProtect Bytecode CFG
Experimental Results

Unpacked real program experiments (a function of NOTEPAD.EXE)

Original x86 CFG

Extracted Bytecode CFG (VMProtect)
Discussion

- **Attacks utilizing alternative emulation approaches**
  - Our loop detection does not work on threaded approaches, but VPC identification would still works
  - Our methods are likewise not applicable to dynamic translation
    - But dynamic translation can be detected using unpacking heuristics

- **Completeness of bytecode syntax and semantics**
  - We might only discover a subset of bytecode instructions
  - Static analysis can be used on central decode dispatch loop

- **Converting byte code to native code**
  - Our extracted semantics can help convert to native code
  - Need multi-path exploration of bytecode instructions to completely convert bytecode program to native code
Thank you

Questions?

msharif@cc.gatech.edu