Static Analysers for Black Hats and White Hats

August 2017
Research Overview

• Decision procedures
• Program analysis
  – Software model checking (CBMC, ...)
  – Concurrency
  – Security
• Software synthesis
• Hardware verification
Oxford: Security Team

John Galea  Sean Heelan  Daniel Neville  Chris Smowton

Will also mention work by Georg Weissenbacher and Matt Lewis
Outline

Black Hats
- Black Hats
- Exemplars for entry points of attacks

A bit of x86
- Registers, memory
- Instructions, IP
- Memory layout, stacks

Symbolic Execution
- Path-based
- Merging paths
- Basic idea of solving the formulas

Automated Black Hats
- Symbolic execution on x86
- Exemplars
- Acceleration
- Outlook: open problems
Black Hat USA 2010: Jackpotting Automated Teller Machines Redux 4/5
107K views

Daniel Kroening, Marktoberdorf 2017
Black Hats

• Obtain control over electronic devices
  – Ideally targeted, but mass market has appeal as well
  – Your phone, laptop, household electronics
  – But could also be your car
  – Or cloud servers
  – Medical records, ...
  – Good exploits are >$1m
WannaCry
WannaCry

• Based on EthernalBlue
• Infection via bug in SMB implementation in Windows (CVE-2017-0144)
• Microsoft: built by NSA
• Leaked 8th of April 2017
• Fix released by Microsoft on 14th of March
Exemplars for Attack Vectors

• Websites
• Any file you open (or preview)
• Software updates
• WiFi, even before connecting
  (remember Iphone baseband bugs)
• 2G, 3G, 4G, no “call” needed, even analog
• USB, Firewire
Brief x86 tutorial

- Intel 8086 / 8088 (1978)
  - 5 – 10 MHz, 16-bit
  - IBM PC
- Intel 80286 (1982)
  - 6 – 12 MHz
  - IBM PC AT
- Intel (80)386 (1985)
  - 16 – 33 MHz, 32-bit
  - PC & IBM PS/2
- Intel 486 (1989)
  - 25 – 100 MHz
  - PC
- Intel Pentium (1993)
  - 60 – 200 MHz
  - PC
x86: a Warning

- Enormously complex ISA
- Thousands of instructions
- Expertise really needs 10+ years
- Many historic artefacts

- Many unique features
- But principles apply to ARM, GPUs, ...
Brief x86 tutorial

CPU

- IP
- eax
- ebx
- ecx
- ZF
- registers

ALU
- Float
- Memory

FUs
- L1, L2
- caches

memory module
memory module
I/O (USB, ...)

data address control


Daniel Kroening, Marktoberdorf 2017
High-level view

- CPU processes a sequential assembler program
- Data held in registers
- Program controls which data is given to which FU, and where the result is stored
- Program controls transfer of data between registers and memory
X86 ISA State

RAM

- Contains data and the program

Data registers

<table>
<thead>
<tr>
<th>Index</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>eax</td>
<td>ecx</td>
<td>edx</td>
<td>ebx</td>
<td>esp</td>
<td>ebp</td>
<td>esi</td>
<td>edi</td>
</tr>
</tbody>
</table>

Instruction Pointer (IP)

- Points to address of current instruction

Flag registers (ZF, ...)

- Store flags for branches
X86 Example Instructions

- **add/sub**: Addition/subtraction of the values in two registers; ZF is set appropriately

- **RRmov**: copies value of one register into another
- **RMmov**: copies value of a register into RAM
- **MRmov**: copies value from RAM into a register

- **jnz**: Jumps to relative address if ZF = 0
X86 Example: add

add eax, edx

- Intel convention: the target register is always on the left-hand side.
- The target register is a source register, too!
- Semantics:

  \[ \text{eax} \leftarrow \text{eax} + \text{edx} \]
X86 Loads and Stores

- Loads and stores have a Displacement:

  \[ ea = esi + \text{Displacement} \]

- The displacement is included in the instruction word as immediate constant

- The register \( esi \) is used as offset
<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Semantics</th>
<th>Opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>RD ← RD+RS</td>
<td>01</td>
</tr>
<tr>
<td>sub</td>
<td>RD ← RD-RS</td>
<td>29</td>
</tr>
<tr>
<td>jnz</td>
<td>if(¬ZF) IP ← IP+Distance</td>
<td>75</td>
</tr>
<tr>
<td>RRmov</td>
<td>RD ← RS</td>
<td>89</td>
</tr>
<tr>
<td>RMmov</td>
<td>MEM[ea] ← RS</td>
<td>89</td>
</tr>
<tr>
<td>MRmov</td>
<td>RS ← MEM[ea]</td>
<td>8b</td>
</tr>
<tr>
<td>hlt</td>
<td></td>
<td>f4</td>
</tr>
</tbody>
</table>
Load Example

\textbf{mov} edx, [\textbf{BYTE} one+esi]

\begin{itemize}
  \item \textbf{Opcode (MRmov)}: 8B 56 17
  \item \textbf{Displacement}: 01 010 110
  \item \textbf{edx}: 110
\end{itemize}

\textbf{Semantics:}

\textbf{edx} ← MEM[esi+17]
# Example Program

<table>
<thead>
<tr>
<th>Address</th>
<th>Machine Code</th>
<th>Assembler using Mnemonics</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>29 F6</td>
<td>sub esi, esi</td>
</tr>
<tr>
<td>02</td>
<td>29C0</td>
<td>sub eax, eax</td>
</tr>
<tr>
<td>04</td>
<td>29DB</td>
<td>sub ebx, ebx</td>
</tr>
<tr>
<td>06</td>
<td>8B 56 17</td>
<td>l mov edx, [BYTE one+esi]</td>
</tr>
<tr>
<td>09</td>
<td>01D0</td>
<td>add eax, edx</td>
</tr>
<tr>
<td>0B</td>
<td>01C3</td>
<td>add ebx, eax</td>
</tr>
<tr>
<td>0D</td>
<td>89C1</td>
<td>mov ecx, eax</td>
</tr>
<tr>
<td>0F</td>
<td>8B 56 1B</td>
<td>mov edx, [BYTE ten+esi]</td>
</tr>
<tr>
<td>12</td>
<td>29D1</td>
<td>sub ecx, edx</td>
</tr>
<tr>
<td>14</td>
<td>75 F0</td>
<td>jnz l</td>
</tr>
<tr>
<td>16</td>
<td>F4</td>
<td>hlt</td>
</tr>
<tr>
<td>17</td>
<td>01 00 0000</td>
<td>one dd 1</td>
</tr>
<tr>
<td>1B</td>
<td>0A 00 0000</td>
<td>ten dd 10</td>
</tr>
</tbody>
</table>