Static Analysis of Executables to Detect Malicious Patterns

[12th USENIX Security Symposium, 2003]

Mihai Cristodorescu
Somesh Jha
CS @ University of Wisconsin, Madison

Presented by K. Vikram
Cornell University
Problem & Motivation...

- Malicious code is ... malicious
- Categorize: Propagation Method & Goal
  - Viruses, worms, trojan horses, spyware, etc.
- Detect Malicious Code
  - In executables
The Classical Stuff

- Focus mostly on Viruses
  - Code to replicate itself + Malicious payload
  - Inserted into executables
- Look for *signatures*
- Not always enough
- Obfuscation-Deobfuscation Game
Common Obfuscation Techniques

- Encryption
- Dead Code insertion*
- Code transposition*
- Instruction Substitution*
- Register reassignment*
- Code Integration
- Entry Point Obscuring
Common Deobfuscation Techniques

- Regular Expressions
- Heuristic Analyses
- Emulation

Mostly Syntactic…
The Game

- Vanilla Virus
- Register Renaming
- Packing/Encryption
- Code Reordering
- Code Integration

- Signatures
- Regex Signatures
- Emulation/Heuristics
- ?
- ?
Current Technology

- Antivirus Software
  - Norton, McAfee, Command
- Brittle
  - Cannot detect simple obfuscations
  - nop-insertion, code transposition
- Chernobyl, z0mbie-6.b, f0sf0r0, Hare
Theoretical Limits

- Virus Detection is undecidable
- Some Static Analyses are undecidable
- But, Obfuscation is also hard
Procedure

Key Ideas:
- Analyze program’s semantic structure
- Use existing static analyses (extensible)
- Use uninterpreted symbols

Abstract Representation of Malicious Code

Abstract Representation of Executable
- Deobfuscation

Detect presence of malicious code
The Annotator

- **Inputs:**
  - CFG of the executable
  - Library of Abstraction Patterns

- **Outputs:**
  - Annotated CFG
Some groundwork

- Instruction $I : \tau_1 \times \ldots \times \tau_k \rightarrow \tau$
- Program $P : \langle I_1, \ldots, I_N \rangle$
- Program counter/point
  - $pc : \{ I_1, \ldots, I_N \} \rightarrow [1, \ldots, N]$
  - $pc(I_j) = j, \forall 1 \leq j \leq N$
- Basic Block, Control Flow Graph*
- Static Analysis Predicates
- Types for data and instructions
### Example Predicates

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dominators($B$)</td>
<td>the set of basic blocks that dominate the basic block $B$</td>
</tr>
<tr>
<td>PostDominator($B$)</td>
<td>the set of basic blocks that are dominated by the basic block $B$</td>
</tr>
<tr>
<td>Pred($B$)</td>
<td>the set of basic blocks that immediately precede $B$</td>
</tr>
<tr>
<td>Succ($B$)</td>
<td>the set of basic blocks that immediately follow $B$</td>
</tr>
<tr>
<td>First($B$)</td>
<td>the first instruction of the basic block $B$</td>
</tr>
<tr>
<td>Last($B$)</td>
<td>the last instruction of the basic block $B$</td>
</tr>
<tr>
<td>Previous($I$)</td>
<td>![Equation](union over $B' \in$ Pred($B_I$) last($B'$) if $I =$ First($B_I$))</td>
</tr>
<tr>
<td>Next($I$)</td>
<td>![Equation](union over $B' \in$ Succ($B_I$) first($B'$) if $I =$ Last($B_I$))</td>
</tr>
<tr>
<td>Kills($p, a$)</td>
<td>true if the instruction at program point $p$ kills variable $a$</td>
</tr>
<tr>
<td>Uses($p, a$)</td>
<td>true if the instruction at program point $p$ uses variable $a$</td>
</tr>
<tr>
<td>Alias($p, x, y$)</td>
<td>true if variable $x$ is an alias for $y$ at program point $p$</td>
</tr>
<tr>
<td>LiveRangeStart($p, a$)</td>
<td>the set of program points that start the $a$’s live range that includes $p$</td>
</tr>
<tr>
<td>LiveRangeEnd($p, a$)</td>
<td>the set of program points that end the $a$’s live range that includes $p$</td>
</tr>
<tr>
<td>Delta($p, m, n$)</td>
<td>the difference between integer variables $m$ and $n$ at program point $p$</td>
</tr>
<tr>
<td>Delta($m, p_1, p_2$)</td>
<td>the change in $m$’s value between program points $p_1$ and $p_2$</td>
</tr>
<tr>
<td>PointsTo($p, x, a$)</td>
<td>true if variable $x$ points to location of $a$ at program point $p$</td>
</tr>
</tbody>
</table>
Abstraction Patterns

- Abstraction pattern $\Gamma : (V,O,C)$
  - $V = \{ x_1 : \tau_1, \ldots, x_k : \tau_k \}$
  - $O = \langle \lambda (v_1, \ldots, v_m) \mid \lambda : \tau_1 \times \ldots \times \tau_m \rightarrow \tau \rangle$
  - $C = \text{boolean expression involving static analysis predicates and logical operators}$

- Represents a deobfuscation
- Predicate controls pattern application
- *Unify* patterns with sequence of instructions
Example of a pattern

$$
\Gamma( X : int(0 : 1 : 31) ) =
\begin{cases}
\{ X : int(0 : 1 : 31) \}, \\
\langle p_1 : \text{"pop } X\text{"}, \\
p_2 : \text{"add } X, 03AFh\text{"} \rangle,
\end{cases}
\quad
p_1 \in \text{LiveRangeStart}(p_2, X)
$$
Defeating Garbage Insertion

<instruction A>
add ebx, 1
sub ebx, 1
nop
<instruction B>

Pattern:
instr 1
...
instr N
Where
Delta(state pre 1, state post N) = 0
Defeating Code-reordering

Pattern:

\[
\text{jmp TARGET} \\
\text{where} \\
\text{Count (CFGPredecessors(TARGET))} = 1
\]
The Annotator

- Given set of patterns $\Sigma = \{ \Gamma_1, \ldots, \Gamma_m \}$
- Given a node $n$ for program point $p$
- Matches each pattern in $\Sigma$ with
  $\langle \ldots, \text{Previous}^2(I_p), \text{Previous}(I_p), I_p \rangle$
- Associates all patterns that match with $n$
- Also stores the bindings from unification
The Detector

- **Inputs:**
  - Annotated CFG for a procedure
  - Malicious code *representation*

- **Output:**
  - Sequence of instructions exhibiting the malicious pattern
Malicious Code Automaton

- Abstraction of the vanilla virus
- 6-tuple \((V, \Sigma, S, \delta, S_0, F)\)
  - \(V = \{ \nu_1: \tau_1, \ldots, \nu_k: \tau_k \}\)
  - \(\Sigma = \{ \Gamma_1, \ldots, \Gamma_n \}\)
  - \(S = \text{finite set of states}\)
  - \(\delta : S \times \Sigma \rightarrow 2^S\) is a transition function
  - \(S_0 \subseteq S\) is a non-empty set of \textit{initial} states
  - \(F \subseteq S\) is a non-empty set of \textit{final} states
Malicious Code

WVCTF:

```
mov   eax, dr1
mov   ebx, [eax+10h]
mov   edi, [eax]
```

LOWVCTF:

```
pop   ecx
jecxz SFMM
mov   esi, ecx
mov   eax, 0d601h
pop   edx
pop   ecx
call  edi
jmp   LOWVCTF
```

SFMM:

```
pop   ebx
pop   eax
stc
pushf
```
Detector Operation

- Inputs:
  - $\text{CFG } P_\Sigma$
  - $\mathcal{A} = (V, \Sigma, S, \delta, S_0, F)$

- Determines whether the same (malicious) pattern occurs both in $\mathcal{A}$ and $\Sigma$

- More formally, tests the emptiness of

  $$L(P_\Sigma) \cap (\bigcup_{B \in B_{\text{All}}} L(B(\mathcal{A})))$$
Detector Algorithm

- Dataflow-like Algorithm
- Maintain a \textit{pre} and \textit{post} list at each node of the CFG \( P_{\Sigma} \)
- List is of \([s, B_s]\), \( s \) is a state in \( A \)
- Join operation is union
Detector Algorithm

- Transfer Function:

  $$\text{foreach } [s, B_s] \in L_n^{pre}$$
  $$\text{foreach } [\Gamma, B] \in \text{Annotation}(n)$$
  $$\land \text{Compatible}(B_s, B)$$
  $$\text{add } [\delta(s, \Gamma), B_s \cup B] \text{ to } NewL_n^{post}$$

- Return:

  $$\exists n \in N . \exists [s, B_s] \in L_n^{post} . s \in F$$
Defenses Against...

- Code Re-ordering
- Register Renaming
- Insertion of irrelevant code
  - nops*, code that modifies dead registers
  - Needs live-range and pointer analyses
Experimental Results

- False Positive Rate : 0
- False Negative Rate : 0
  - not all obfuscations are detected
Performance

![Bar chart showing time (seconds) for different applications.]

- **tiffdither** (6656): 7.363 seconds
- **winmine** (12120): 17.950 seconds
- **spyxx** (307200): 224.584 seconds
- **QuickTimePlayer** (499712): 959.913 seconds

Legend:
- Detector avg.
- Annotator avg.
Future Directions

- **New languages**
  - Scripts – VB, JavaScript, ASP
  - Multi-language malicious code

- **Attack Diversity**
  - Worms, trojans too

- **Irrelevant sequence detection**
  - Theorem provers

- **Use TAL/external type annotations**
Pitfalls/Criticisms?

- Focus on viruses instead of worms
- Still fairly Ad-hoc
- Treatment of obfuscation is not formal enough
- Intractable techniques
  - Use of theorem provers to find irrelevant code
- Slow
- No downloadable code
- Not enough experimental evaluation