Impeding Malware Analysis Using Conditional Code Obfuscation

Monirul Sharif¹, Andrea Lanzi², Jonathon Giffin¹, Wenke Lee¹

¹Georgia Institute of Technology
²Università degli Studi di Milano

NDSS 2008
Introduction

We need to understand malware...

Malware

Exploits

Propagation Control Capabilities

System-wide effects

Reverse engineering and Malware Analysis

Hundreds of new malware samples appear almost everyday...

Malware

Malware

Malware

Malware

Malware

Malware

Automated analysis systems have become very important

Obfuscations that are easily applicable on existing code can be a threat

We present a **Simple, Automated** and **Transparent** Obfuscation against state-of-the-art malware analyzers
Malware Analysis and Obfuscations

**Defense**

Static Analysis based approaches

Dynamic malware analysis

Dynamic multipath exploration (Moser et al. 2007)

Bitscope (Brumley et al. 2007)

EXE (Cadar et al. 2006)

Forced execution (Wilhelm et al. 2007)

**Offense**

Polymorphism, metamorphism, packing, opaque predicates, anti-disassembly

Trigger-based behavior (Logic bombs, time bombs, anti-debugging, anti-emulation, etc.)

? Conditional Code Obfuscation
Rest of the Talk

- **Conditional Code Obfuscation**
  - Principles
  - Static analysis based automation
  - Automatic applicability on existing malware without modification

- **Implications**
  - Implications on Existing Analyzers
  - Measuring Obfuscation Strength

- **Prototype Implementation and Evaluation**
  - Evaluation on malware

- **Weaknesses and Defense**
  - How analysis can be improved to defender
Principles of Our Attack

- Malware Binary
- Condition
- Trigger-based behavior

Inputs
- Unknown

Any static and dynamic analysis approach

Input Oblivious Analyzer
Impeding Malware Analysis Using Conditional Code Obfuscation

**Principles of Our Attack**

```
Malware Binary
  Condition
  Trigger-based behavior
```

```
cmd = get_command(sock);
if (strcmp(cmd, "logkeys") == 0))
{
    LogKeys()
}
```

```
cmd = get_command(sock);
if (Hash(cmd) == H)
{
    LogKeys()
}
```
Principles of Our Attack

---

**Malware Binary**

- **Unknown Inputs**
- **Condition**
- **Trigger-based behavior \((K)\)**

---

```
cmd = get_command(sock);
if (strcmp(cmd, "logkeys") == 0))
{
    LogKeys()
}
```

```
encr_LogKeys()
```

---

```
cmd = get_command(sock);
if (strcmp(cmd, "logkeys") == 0))
{
    decrypt(enr_LogKeys, K);
    enr_LogKeys()
}
```

---

```
encr_LogKeys()
```
Impeding Malware Analysis Using Conditional Code Obfuscation

Principles of Our Attack

Malware Binary

Unknown Inputs

Condition

Trigger-based behavior

The key is no longer inside the code

```
cmd = get_command(sock);
if (strcmp(cmd, "logkeys") == 0))
{
    LogKeys()
}
```

```
cmd = get_command(sock);
if (Hash(cmd) == H))
{
    decrypt(encr_LogKeys, cmd);
    encr_LogKeys();
}
```

```
encri_LogKeys();
```
**General Obfuscation Mechanism**

**Original Code**

```c
if ( X == c ) {
    B
}
```

**Obfuscated Code**

```c
if ( Hash(X) == Hc ) {
    Decr(B_E, X)
    Encri(B, c)
}
```

- **Candidate Conditions - Conditions with equality**
  - Hash function Properties:
    - The usual `==` operator
    - Pre-image resistance – Protects against reversing
      Hard to find c given H_c
    - String equality checks – strcmp, memcmp, strncmp etc.
    - Conditions with ‘>’, ‘<’, ‘!=’ will not work
  - **Conditional Code**
    - Second pre-image resistance - Program correctness
      Hard to find another c where Hash(c) = H_c
Automation Using Static Analysis

- Identify Candidate Conditions
  - Identify functions and create CFG for each function
  - Find blocks containing candidate conditions

- Conditional code Identification
  - Intra-procedural - Basic blocks control dependent on condition with true outcome
  - Inter-procedural - Set of all functions only reachable from selected basic blocks

- Exclude functions reachable from default path
  - Conservative conditional code selection for function pointers
Two keys are used in two paths. Duplicate code.
If one path is not candidate condition, no use in concealing the trigger code.
Handling Complex Conditions

```java
if ( X==a && Y==b ) {
    Attack()
}
```

- Logical “and”

```java
if ( X==a ) {
    if (Y==b ) {
        Attack()
    }
}
```

```java
else if (Y==b ) {
    Attack()
}
```

- Logical “or”

```java
if ( X==a || Y==b ) {
    Attack()
}
```

```java
if ( X==a ) {
    if (Y==b ) {
        Attack()
    }
}
else if (Y==b ) {
    Attack()
}
```
Handling Complex Conditions

```c
switch (cmd) {
    case 0:
        attack1();
        break;
    case 1:
        recon();
        attack2();
    case 2:
        attack2();
}
```

Switch Case
Consequences to Existing Analyzers

- Multi-Path Exploration (Moser et al., Bitscope)
  - Constraints are built for each path
  - Hash functions are non-linear, so cannot find solution

- Input Discovery (EXE)
  - Solves constraints to get inputs – symbolic execution
  - Same problem, cannot find derive input

\[ \text{Hash}(X) = H_C \]

Trigger-based behavior
Consequences to Existing Analyzers

- **Forced Execution**
  - Without solving constraints, forces execution
  - Without key, program crashes

- **Static Analysis**
  - Same as packed code, static analysis on trigger code is not possible

```plaintext
Hash(X) == H_c
```

Trigger-based behavior
Attacks on the Obfuscation

- Attacks on Hash(X) = H_c
  - Find possible X for satisfying the above

- Input domain
  - Domain(X) – set of all possible values X may take
  - With time $t$ for every hash computation,
    total time = Domain(X)$t$
  - For an integer I, Domain(I) = $2^{32}$

- Brute Force attacks

- Dictionary Attacks
Prototype Implementation

• Overview
  o Implemented for Linux
  o Takes malware C source code and outputs obfuscated ELF binaries

• Analysis Level – both source code and binary levels required
  o Source and IR level – *type information is essential*
  o Binary level – *decrypted code must be executable*

Simplified architectural view of the automated obfuscation system
Analysis and Transformation Phase

- Candidate Code Replacement
  - Enc(X)/Dec(X) Encryption/Decryption – AES with 256 bit keys
  - Hash function – Hash(X) - SHA-256
  - Different hash functions based on data type of X
- Decryption Keys and Markers
  - Key generation – Key(X) = Hash(X|N), N is Nonce
Encryption Phase

- DynInst based binary transformation tool
  - Finds Decipher(), and End_marker() and key (K_c)
  - Encrypts binary code with key
  - Removes marker and key from code
Experimental Results

• Evaluated by Obfuscating Malware Programs
  o Selected representative malware source programs for Linux with trigger based behavior

• Evaluation Method
  o Manually identified malicious triggers in malware
  o Applied obfuscation, counted how many were completely obfuscated by the automated system
  o Considered three levels of obfuscation strength –
    Strong – strings
    Medium – integers
    Weak – booleans and return codes
## Experimental Results

<table>
<thead>
<tr>
<th>Malware</th>
<th>Candidate Conditions</th>
<th>Malicious Triggers</th>
<th>Strong</th>
<th>Medium</th>
<th>Weak</th>
<th>None</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slapper Worm (P2P Engine)</td>
<td>157</td>
<td>28</td>
<td>-</td>
<td>28</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Slapper Worm (Backdoor)</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>BotNET (IRC Botnet server)</td>
<td>61</td>
<td>52</td>
<td>52</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>passwd rootkit</td>
<td>5</td>
<td>2</td>
<td>2</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>login rootkit</td>
<td>19</td>
<td>3</td>
<td>2</td>
<td>-</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>top rootkit</td>
<td>17</td>
<td>2</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>2</td>
</tr>
<tr>
<td>chsh rootkit</td>
<td>10</td>
<td>4</td>
<td>2</td>
<td>-</td>
<td>2</td>
<td>-</td>
</tr>
</tbody>
</table>
Analysis of the Technique (Strengths and Weaknesses)

- Knowledgable attacker can modify program to improve obfuscation effectiveness
  - Increase candidate conditions - replace <, >, != operators with ‘==’
  - Increase conditional code – incorporate triggers that encapsulate more execution behavior
  - Increase input domains - Use variables with larger domains (e.g. strings) or use larger integers

- Weaknesses
  - Input domain may be very small in some cases
  - Upside on Malware detection – but polymorphic layers can be added
Defense Approaches

- **Incorporating cracking engine**
  - Equipped with decryptors where various keys are tried out repeatedly
  - Input domain knowledge (for dictionary attacks)
    - Determine type information – reduce domain space
    - Syscall return codes

- **Input-aware analysis**
  - Gather I/O traces along with malware binaries
Conclusion

- We presented an obfuscation technique that can be widely applicable on existing malware.
- The obfuscation conceals trigger based behavior from existing and future analyzers.
- We have shown its effectiveness on malware using our implemented automated prototype.
- We presented its weaknesses and possible ways analyzers can be improved to defeat it.
Questions?

msharif@cc.gatech.edu
## Experimental Results

<table>
<thead>
<tr>
<th>Malware</th>
<th>Candidate Conditions</th>
<th>Original Size</th>
<th>Obfuscated Size</th>
<th>% size increase</th>
</tr>
</thead>
<tbody>
<tr>
<td>Slapper Worm (P2P Engine)</td>
<td>157</td>
<td>82.8 KB</td>
<td>97.3 KB</td>
<td>17%</td>
</tr>
<tr>
<td>Slapper Worm (Backdoor)</td>
<td>2</td>
<td>3.3 KB</td>
<td>10.7 KB</td>
<td>224%</td>
</tr>
<tr>
<td>BotNET (IRC Botnet server)</td>
<td>61</td>
<td>100.8 KB</td>
<td>115.1 KB</td>
<td>14%</td>
</tr>
<tr>
<td>passwd rootkit</td>
<td>5</td>
<td>6.9 KB</td>
<td>13.8 KB</td>
<td>172%</td>
</tr>
<tr>
<td>login rootkit</td>
<td>19</td>
<td>19.2 KB</td>
<td>27.3 KB</td>
<td>42%</td>
</tr>
<tr>
<td>top rootkit</td>
<td>17</td>
<td>43.9 KB</td>
<td>53.6 KB</td>
<td>22%</td>
</tr>
<tr>
<td>chsh rootkit</td>
<td>10</td>
<td>6.9 KB</td>
<td>14.3 KB</td>
<td>107%</td>
</tr>
</tbody>
</table>