Practical Techniques for Regeneration and Immunization of COTS Applications

Lixin Li  
Mark R. Cornwell  
E. Hultman  
James E. Just  
R. Sekar

Global InfoTek, Inc  
Stony Brook University

(Research supported by DARPA, NSF and ONR)

Problem

- Windows of vulnerability in a production system
  - Zero-day vulnerability is a serious concern
  - Exploits produce faster than patches
  - End users take time to deploy patches

- Ideal solution requires:
  - Defends against most common attacks
  - Protects application integrity as well as availability
  - Low performance overhead
  - Working on COTS
Related Work

- Taint-tracking:
  - Drawbacks
    - Intrusive instrumentation and
    - High overhead and
    - Somewhat language-specific

- Automatic Signature Generation:
  - Drawbacks
    - Heavy weight or
    - Requires accurate attack replay

Our Research

- Approach applicable to most popular attacks, from memory corruption attacks to string injection attacks
- Our project, RAMSES (Regeneration and Immunization Services), provides regeneration and immunization services
  - Regeneration provided by our ASR implementation on Windows binary (implemented prior to Windows Vista)
  - Immunizes systems against zero-day attacks, preserve integrity and availability
    - Inspired by biological immune system to learn from attacks
    - Provides much targeted and effective response without damaging host
- Protect against attack variants and brute-force attacks
  - Vulnerability-oriented signature generation
### Attack Space of Interest (CVE 2008-09)

- **Stack Overflow**: 14%
- **Heap Overflow**: 3%
- **Format String**: 1%
- **SQL Injection**: 23%
- **Cross-Site Scripting**: 16%
- **Directory Traversal**: 8%
- **Other Injection**: 1%
- **Others**: 34%

### Attack Scenarios

- **String injection vs Memory corruption attacks**
  - **String injection attacks**:
    - Target output requests to access protected services (SQL database), subsystems (command interpreters)
  - **Memory corruption attacks**:
    - Target critical data structures (stack, heap,...)
    - Output interface is not clearly defined
    - Attacker intends to hijack execution to run attacker provided arbitrary code
  - **Attack**: use maliciously crafted input to exert **unintended control** over output operations
  - **Detect “exertion of control”**
    - Output depends on input (taint)
  - **Detect the degree of intended control** is necessary for string injection.
Approach Overview

- **Taint inference:** an efficient and non-invasive alternative to taint analysis
  - Analyze observed inputs and outputs
  - Standard library API Interception is mostly needed
  - Language-neutral
- **Attack detection:** Use taint-inference to correlate attack with causative input
  - Use Address-Space Randomization (ASR) to detect memory corruption attacks
  - Use memory analysis on corrupted areas for attacks detected by ASR
  - Use syntax and taint-aware policies to detect string injections
  - Leverage interplay between taint and structural changes to output requests
- **Immunization:** filter out future attack instances
  - Input Filter
  - Output Filter

Taint Inference

- **Infer taint flow by observing inputs and outputs**
  - Assumption:
    - No arbitrary transformation of Input
    - Standard transformation like encoding can be accommodated
  - Key observation:
    - **Major** parts of input appear at output with **minor** changes
      - Vulnerable buffer corrupted after a buffer overflow and before the attack is detected (taint “untainted”)
      - Simple transformations are common in web applications
  - Solution: use approximate substring matching
    - Given an input $I$ and an output $O$, substring $o$ of output $O$ contains data from input $I$ if there is an approximate string match between $I$ and substring $o$ (the edit-distance between $I$ and $o$ is less than a given threshold)
  - Standard approximate substring matching algorithms have quadratic time and space complexity
  - Our fast approximate substring match algorithm has linear-time complexity
System Components

Interceptors

Protected System

Web Server (IIS/Apache)

Web App (PHP/Java/C++)

Database/Backend Server

System Libraries

RAMSES Components

Taint Inference
- Input parsing
- Output analysis

Attack Detector
- Address-space randomization (ASR)
- Syntax and taint-aware policy

Filter Generator
- Input filter
- Output filter

System Architecture

RAMSES Interceptor

File Interfaces

Network Interfaces

OS & Application Specific Interfaces

Plug-in Parser/Logger (Application Configurable)

File Format Parser Plug-In

Network Protocol Parser Plug-In

OS & App. Specific Plug-In

In-memory Parsed Content

RAMSES Attack Detection & Analysis

ASR & Crash Analysis

Syntax and Taint-Aware Policy

Response Generation

Output Filter Generation

Taint-based Correlation

Signature Generation

Signature Refinement
Lexical and Syntactic Confinement

- Basis of an uniformed approach to detect both memory corruption attacks and string injection attacks
  - Key observation:
    - Attack-bearing input breaks output structures
- String injection attacks
  - Lexical confinement
    - Tainted data should not flow past the end of tokens in output parse tree
  - Syntactic confinement
    - Tainted data should not straddle two sub-trees of the output parse tree
- Memory corruption attacks
  - Lexical/Syntactic confinement generally applicable
  - Buffer overflow breaks local array confinement and break stack/heap structures

Illustration of Lexical/Syntactic Confinement

Benign Input: xyz@abc.com  Attack Input: nobody; rm –rf *

```
Stack frame 1
  Return Address
Stack frame 2
  Return Address
Stack frame 2
```
**Stack Frame Structure**
- Windows Stack Structure

```
High

EBP
FS:0
ESP

Low

A's stack frame
Function arguments
Return address

Previous stack frame
Exception Registration Record
Local variables

B's stack frame
Function arguments
Return address (to A)

Previous stack frame
Local variables

C's stack frame
Function arguments
Return address (to B)

Previous stack frame
Exception Registration Record
Local variables
```

**Heap Block Structure**

<table>
<thead>
<tr>
<th>Size</th>
<th>Previous Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Segment Index</td>
<td>Flags</td>
</tr>
<tr>
<td></td>
<td>FLink</td>
</tr>
<tr>
<td></td>
<td>BLink</td>
</tr>
</tbody>
</table>

Windows Free Heap Block Structure
Memory Security Analysis Steps

- **Input Parsing:**
  - Parsing input into meaningful (field, value) pairs
  - To simplify taint-inference
  - Provide semantic context for signature generation

- **Crash exception analysis**
  - Corruption Area and Targets Analysis
  - Buffer Confinement Analysis
    - Taint inference
    - Approximate string match naturally handles “untaint” after buffer overflow

- **Signature generation**
  - Length-based
  - Apply at the right semantic context

---

Memory Security Analysis & Implementation

[Diagram showing the integration of components such as Exception, Input parsing, Signature, RAMSES IMMUNIZER, RAMSES Crash Monitor, RAMSES Crash Analyzer, and Signature Generator.]

- **ASR**
  - Protected Application
  - Ramses Interceptor

- **Uses**
  - Microsoft Detour
  - Windows Debug API

- **RAMSES IMMUNIZER**
  - Crash Monitor
    - Catch security events (exceptions)
  - Crash Analyzer
    - Taint-inference
    - Approximate matching
  - Signature Generator
    - Input correlation, Signature refinement
Illustration of taint infer a stack buffer overflow

1. pBuffer2[32] is a vulnerable buffer, input overflow pBuffer2, another local buffer pBuffer1[32], Exception Handler, Saved EBP, return address and beyond.
   - pBuffer1[32] is “untainted” before attack is detected
2. Crash security analysis identified corrupted area and interesting targets
3. Apply approximate string match on input value from input parsing step and corrupted area to identify starting boundary and ending boundary of the vulnerable buffer.
   - Despite the “gap” caused by “untaint”, approximate string match succeeds.
Filters

- Input filters block attack input from passing through input functions
  - Return error code for the input function
  - Servers written to expect network errors
- Input filter is used for memory corruption attacks
  - Output interface is not clearly defined
  - Output filter is usually too late
    - Output is usually in the form of program crash
- Output filter is used for string injection attacks
  - Block policy-violating outputs from carrying out output functions
  - Intercept the output function and returning an error code
  - Recovery is usually possible for string injection attacks

Preliminary Results (cont)

- Input filter: Memory corruption attacks

<table>
<thead>
<tr>
<th>Vulnerability Type</th>
<th>Attack Target</th>
<th>Payload Type</th>
<th>Attack Description</th>
<th>Vulnerable Buffer Identified</th>
<th>Blocking Signature Generated</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synthetic</td>
<td>MEVS</td>
<td>Working exploit</td>
<td>Stack buffer overflow to overwrite return address</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Synthetic</td>
<td>MEVS</td>
<td>DDoS</td>
<td>Stack buffer overflow to overwrite return address</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Synthetic</td>
<td>MEVS</td>
<td>Working exploit</td>
<td>Stack buffer overflow to overwrite SDH</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Synthetic</td>
<td>KE-GAPI extension DLL</td>
<td>Working exploit with payload encoded</td>
<td>Stack buffer overflow to overwrite SDH</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Synthetic</td>
<td>MEVS</td>
<td>Working exploit</td>
<td>Heap Overflow Freelist[00] to trigger double pointer unlink</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Synthetic</td>
<td>MEVS</td>
<td>DDoS</td>
<td>Heap Overflow [1 - 127] to trigger double pointer unlink</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Synthetic</td>
<td>MEVS</td>
<td>Working exploit</td>
<td>Heap Overflow Lookaside list to trigger double pointer unlink</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Synthetic</td>
<td>MEVS</td>
<td>DDoS</td>
<td>Heap Overflow triggering block coherence and double pointer unlink</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Synthetic</td>
<td>MEVS</td>
<td>DDoS</td>
<td>Heap overflow with original input reversed before overflow</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Real world (CVE-2004-1134)</td>
<td>IIS</td>
<td>Working exploit</td>
<td>Overflow a stack buffer in whoami.dll to overwrite SEH</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Real world (CVE-2008-20909)</td>
<td>FreeFTPd 1.08</td>
<td>Working exploit</td>
<td>Overflow a stack buffer in freeFTPd service to overwrite SEH</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Preliminary Results

- Output filter: Web Applications

<table>
<thead>
<tr>
<th>Application</th>
<th>Language</th>
<th>Size (lines)</th>
<th>Environment</th>
<th>Attacks</th>
<th>Comments</th>
<th>Detection</th>
<th>False Positive</th>
</tr>
</thead>
<tbody>
<tr>
<td>phpBB 2.0.5</td>
<td>PHP/C</td>
<td>34K</td>
<td>IIS, Apache</td>
<td>SQL injection</td>
<td>CAN-2003-0486</td>
<td>Yes</td>
<td>None</td>
</tr>
<tr>
<td>SquirrelMail 1.4.0</td>
<td>PHP/C</td>
<td>42K</td>
<td>IIS, Apache</td>
<td>Shell cmd injection</td>
<td>CAN-2003-0990</td>
<td>Yes</td>
<td>None</td>
</tr>
<tr>
<td>SquirrelMail 1.2.10</td>
<td>PHP/C</td>
<td>35K</td>
<td>IIS, Apache</td>
<td>XSS</td>
<td>CAN-2002-1341</td>
<td>Yes</td>
<td>None</td>
</tr>
<tr>
<td>PHP/ XMLRPC</td>
<td>PHP/C</td>
<td>2K</td>
<td>IIS, Apache</td>
<td>PHP cmd injection</td>
<td>CAN-2005-1921</td>
<td>Yes</td>
<td>None</td>
</tr>
<tr>
<td>ANNEXIA (5 apps)</td>
<td>Java/C</td>
<td>30K</td>
<td>Apache/Tomcat</td>
<td>SQL injection</td>
<td>21K attacks, 3.8K legitimate</td>
<td>100%</td>
<td>0%</td>
</tr>
<tr>
<td>WebGoat</td>
<td>Java</td>
<td>Tomcat</td>
<td>HTTP response splitting Shell cmd injection</td>
<td>Yes</td>
<td>Yes</td>
<td>None</td>
<td></td>
</tr>
</tbody>
</table>

End-to-end Performance Overhead

- Memory signature generation time only incurs when exception happens, usually less than 250 ms cpu time for stack overflow and less than 800 ms for heap overflow
- Web Application Performance Overhead

<table>
<thead>
<tr>
<th>Application</th>
<th>Size (LOC)</th>
<th># of Requests</th>
<th>Response time (sec)</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bookstore</td>
<td>9552</td>
<td>605</td>
<td>20.7</td>
<td>1.7%</td>
</tr>
<tr>
<td>Empldir</td>
<td>3028</td>
<td>660</td>
<td>17.3</td>
<td>3.4%</td>
</tr>
<tr>
<td>Portal</td>
<td>8775</td>
<td>1080</td>
<td>31.7</td>
<td>5.1%</td>
</tr>
<tr>
<td>Classifieds</td>
<td>5726</td>
<td>576</td>
<td>18.0</td>
<td>4.3%</td>
</tr>
<tr>
<td>Events</td>
<td>3805</td>
<td>900</td>
<td>23.0</td>
<td>3.1%</td>
</tr>
<tr>
<td>Total</td>
<td>30886</td>
<td>3821</td>
<td>110.7</td>
<td>3.5%</td>
</tr>
</tbody>
</table>
Contact

• Lixin Li, Principal Scientist
  Global InfoTek, Inc., Reston, VA
  • nli@globalinfotek.com

• James E. Just, Director of Research
  Global InfoTek, Inc., Reston, VA
  • jjust@globalinfotek.com

• R. Sekar, Professor of Univ. Stony Brook
  • sekar@cs.stonybrook.edu

Questions?