Review and Analysis of Synthetic Diversity for Breaking Monocultures

James E. Just  
jjust@globalinfotek.com  
Mark W. Cornwell  
mcornwell@globalinfotek.com  
WORM-04  
29 October 2004
Overview

• Problem space
• “Breaking the V-Spec”
• Literature on code transformations
• Effectiveness of transforms
• Functional architecture
• Example
• Conclusions
Problem Space (1):
Excessive Homogeneity => Systemic Vulnerability

- Homogenously vulnerable host population
- Attack finds rich environment to exploit and spread
- Exponential growth leads to catastrophic failure

How to prevent exponentially cascading failures?
Problem Space (2):
Difficulty of Building Intrusion Tolerant Systems

• Intrusion tolerant systems difficult to build
  – Expensive, large hw/sw footprints, \textit{a priori} knowledge of attack modalities
  – Need significant diversity of spares -- even if intrusion tolerance is affordable approach, practical limits exist on diversity of spares
    • N-version programming is costly
    • Limited commercial diversity of similar apps

• Foreseeable cyber-risks dominated by static, durable executable monoculture
Current Attack Problem

Known V-Spec

Vulnerability

Attack

Software Specification

Source Code

Executable Code

Machine-level Code

Linker Loader

Machine Code Specification
Breaking the V-Spec

V-Spec Unknown Until Load-Time
Illustrative Common Techniques & Assumptions
Segment of Code Red 1 Disassembly

1. Relative Virtual Addressing is Windows name for identifying specific memory addresses in executable files without hardcoding (offset to virtual memory load location).

2. This calls specific API (GetProcAddress) for DLL via IAT. GetProcAddress is one of two key functions for the exploit. LoadLibraryA (not shown) is the other.

3. After this, GetProcAddress and LoadLibraryA are used to load kernel32.DLL, infocomm.DLL and WS2_32.DLL to access the file system, open network sockets and send and receive network packets.

---

01 loc_4B4:    ; CODE XREF: DO_RVA+26Dj
02    mov    esi, esp
03    mov    ecx, [ebp-198h]; set ecx with the data segment
04    push   ecx; push data segment (pointer of function to load)
05    mov    edx, [ebp-1CCh]; get current RVA base offset
06    push   edx; push module handle(base loaded address)
07    call   dword ptr [ebp-190h]; call GetProcAddress
Many Assumptions Made

• Breaking the assumptions mechanically
  – Re-arrange the run-time stack
  – Permute the addresses in the jump table
  – Change the machine code (table transformation)
  – Change the interpretation of (encrypt) filenames
  – Change the order of parameters for system calls
  – Encrypt file name parameters to system calls
  – Rename ports for network connections
  – Put return pointers on a separate stack
CMU Ballista Study

• Most production quality operating system and core library code exhibit large numbers of flaws in validating input, call order, etc.

• Specification-driven testing verifies this result.
SANS Top 10 Top Vulnerabilities to Windows Systems

- W1 Web Servers & Services
- W2 Workstation Service
- W3 Windows Remote Access Services
- W4 Microsoft SQL Server (MSSQL)
- W5 Windows Authentication
- W6 Web Browsers
- W7 File-Sharing Applications
- W8 LSAS Exposures
- W9 Mail Client
- W10 Instant Messaging
SANS Top Vulnerabilities to UNIX Systems

- U1 BIND Domain Name System
- U2 Web Server
- U3 Authentication
- U4 Version Control Systems
- U5 Mail Transport Service
- U6 Simple Network Management Protocol (SNMP)
- U7 Open Secure Sockets Layer (SSL)
- U8 Misconfiguration of Enterprise Services NIS/NFS
- U9 Databases
- U10 Kernel
Synthetic Diversity Approaches

• Source code:
  – Easiest area to introduce diversity via compilers
  – Complicates distribution and updating
  – Significant market penetration problems with software vendors

• Binary (executable) code:
  – More difficult to introduce diversity
    • Disassembly is not exact science
    • Runtime and randomized modification is harder still
  – Can be implemented by organizations and users who need the benefits

• Binary code with annotations (hints)
Rich Literature

• Vulnerability descriptions and explanations
• Exploiting vulnerabilities
• Vulnerability and attack taxonomies
• Obfuscation to protect secrets
  – Executing code
  – Code and digital content
• Transforms to mitigate vulnerabilities
  – Source code
  – Executables
Transform Techniques in Literature*

• Obfuscation
  – Layout obfuscation (scramble identifiers, remove comments, change formats)
  – Control flow obfuscations (Statement grouping, ordering, computation, opaque constructs)
  – Data obfuscation (Storage, encoding, grouping, ordering)
  – Preventative transformations (prevent decompilers from operating by exploiting weaknesses)
    • Inherent (aliases, variable or bogus dependencies, opaqueness side effect & difficulty)
    • Targeted

• Source code
  – N-version programming
  – Functional-behavior preserving diversity in components used (e.g., different encryption algorithms, different scales for data such as Celsius or Fahrenheit)
  – Semantics preserving source code transformations
    • Place sensitive data (such as function and data pointer) below the starting address of any buffer
    • Variable ordering
    • Equivalent instructions
  – Variable compilation --Variable internal names, padding and addresses, linking orders
  – Insertion of opaque constructs or other dead code to change memory layout

• Binary code
  – Address transformations (relative and absolute) on binary code
    • Randomize base address of memory regions (Stack, Heap, DLL, routines/static data in executable)
    • Permute order of variable/routines (Local variables in stack frame, static variables, routines in shared libraries or routines in executables)
    • Introduce random gaps between objects (Padding in stack frames, between successive malloc allocation requests, between variables in the static area; Gaps within routines and add jump instructions to skip over gaps)
  – System resource, system call, or DLL name/address transformation
  – Instruction set transformation

*References shown on later slides
Collberg, Thomborson, Low

- First systematic studies of Java code obfuscation
  - Produced taxonomy (layout, control flow, data, and preventative transforms)
  - Low-cost, stealthy opaque constructs
  - Techniques for obscuring data structures and abstractions
  - Measured effectiveness using software complexity metrics
Wang

- Studied malicious host problem to protect trusted probe communicating with trusted host
  - Key threats: impersonation, intelligent tampering, input spoofing, not DOS or random tampering
  - Input spoofing, in general, unsolvable but
    “If spoofing input x requires solving the algorithm-secrecy or execution-integrity problem, then techniques to ensure the later can be used to counteract input spoofing. However, there are applications where this is not possible.”
  - Pervasive aliasing enabled proof: precise analysis of transformed program (e.g., CFG) is NP hard
  - Replacing 50% of branches =>
    - Execution time = 4X
    - Size = 2X

- Wroblewski extended ideas and implemented purely sequential, controllable approach that worked on binary code
Linn and Debray

• Rewrote binaries (IA-86) to disrupt major static disassembly approaches (linear sweep and recursive traversal)
  – Best commercial tools failed on 65% of instructions and 85% of functions
  – Execution times = 1.13 X
  – Executable size = 1.15-1.20 X
Barak et al.

• Seminal proof showed impossibility of completely obscuring code and no general obfuscator possible
  – Badger et al. began to extend Wang’s work but unable to prove minimum resistance time to reverse engineering effort – redirected to review obfuscation work (*tour de force*)
Digital Rights Management

- Malicious host is key problem in DRM
- White box cryptography approach
- Chow et al.
  - Notwithstanding Barak, can provide useful commercial levels of security
  - Obscured DES and AES algorithms
- Jacobs et al.
  - Broke obscured DES but showed general problem of retrieving data from circuits is NP hard
  - Admitted that, in practice, usually easy
- Link and Neumann improved on Chow
Mitigating Vulnerabilities in Code

• Forrest et al. randomized stack resident data addresses via modified gcc compiler
• Chew and Song randomized stack base address, system call numbers & library entry points via modifying Linux loader and kernel system call table and binary rewriting
• Xu et al. modified Linux kernel to randomize base addresses of program regions
• Approaches still vulnerable to relative address attacks
Forrest et al.

• Scrambled executable (prn), then unscrambled through modified code emulator (x86)
  – Speed = 1.05 X
  – Memory usage = 3 X
  – Discussed danger of generating valid instruction during scrambling but did not see experimentally

• Kc produced similar results
Bhaktar et al.

- Focused on memory error exploits
  - Randomized absolute/relative addresses in Linux binary code
  - Approach offered protection against classic attacks
    - Stack smashing, existing code exploits, format string, data modification, heap overflow, double-free, integer overflows
    - Data modification attacks still possible but Etoh and Yoda approach could help

“Key difference between program obfuscation and address obfuscation is that program obfuscation is oriented towards preventing most static analyses of a program, while address obfuscation has a more limited goal of making it impossible to predict the relative or absolute addresses of program code and data. Other analyses, including reverse compilation, extraction of flow graphs, etc., are generally not affected by address obfuscation”
## Performance of Bhaktar Transforms

<table>
<thead>
<tr>
<th>Program</th>
<th>Combination (1)</th>
<th>Combination (2)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>% Overhead</td>
<td>Standard Deviation (% of mean)</td>
</tr>
<tr>
<td>tar</td>
<td>-1</td>
<td>3.4</td>
</tr>
<tr>
<td>wu-ftpdr</td>
<td>0</td>
<td>1.4</td>
</tr>
<tr>
<td>gv</td>
<td>0</td>
<td>6.1</td>
</tr>
<tr>
<td>bison</td>
<td>1</td>
<td>2.0</td>
</tr>
<tr>
<td>groff</td>
<td>-1</td>
<td>1.1</td>
</tr>
<tr>
<td>gzip</td>
<td>-1</td>
<td>1.9</td>
</tr>
<tr>
<td>gnuplot</td>
<td>0</td>
<td>0.9</td>
</tr>
</tbody>
</table>

Combination 1: link time static relocation of stack, heap and code regions with random gaps in stack frames;
Combination 2: load time dynamic relocation of above
# Effectiveness of Transforms

Diversity is not a panacea for achieving cybersecurity. There are many other ways to penetrate a system.

- **Improper Validation**
  - Memory error: Code injection attacks (stack, heap, and integer overflows) + + + + + * <1> ++ ++ ++ ++ ● ● ●
  - Existing code exploits (printf) + + + + ● ● <1> ++ ++ ++ ++ ● ● ●

- **Design errors**
  - Code injection attacks (SQL injection, cross site scripting) + + ● ● ● ● ● ● ● ● + ● ● ●
  - Improper exception handling + ● ● ● ● ● ● ● ● + + ●
  - Protocol errors (IP session hijacking, Unicode attacks, ARP cache poisoning) + ● ● ● ● ● ● ● + + +

- **Improper Exposure**
  - System misconfiguration, e.g., default passwords and accounts + ● ● ● ● ● ● ● ● ● ● ● ● ●
  - Race condition + ● ● ● ● ● ● ● ● + + +
  - Address or data leakage + ● ● ● ● + + ● ● ● ● + + +
  - Social engineering, dumpster diving ● ● ● ● ● ● ● ● ● ● ● ● ●
  - Weak passwords ● ● ● ● ● ● ● ● ● ● ● ● ●
  - Weakly encrypted passwords + ● ● ● ● ● ● ● ● ● ● ● ● ●
  - Memory leakage + ● ● ● ● ● ● ● ● ● ● ● ● ●
  - Resource exhaustion + ● ● ● ● ● ● ● ● ● ● ● ● ●

**Key:** Effectiveness in preventing attack: ++ = highly effective, + = effective, * = negligible, < = potentially counterproductive.
Diversity System Architecture Overview

Input to Policy

Policy → Key Generator

Key Generator → Monitor

Monitor → Execution Space

Execution Space

Input to Original Program

Original Program → Analyzer

Analyzer → Transformer

Transformer → Key

Key → Transformed Code

Transformed Code

Output to Wrapper

Wrapper → Loader

Loader
Diversity System Functional Architecture: Normal

Modified loader transforms original stored program and generates wrapper that retranslates external calls.

Response to normal user inputs are translated & untranslated so they work.

Original Program

Modified Loader

User Inputs

Untranslation

Wrapper

Transformed In-memory program

Annotation File

Random Key

Other System Resources
Diversity System Functional Architecture: Initial Exploit

Modified loader transforms original stored program and generates wrapper that retranslates external calls.

Attacker

User Inputs

Untranslation

Wrapper

Original Program

Modified Loader

Annotation File

Random Key

Transformed In-memory program

Some attacks fail because assumed vulnerability is gone

Response to normal user inputs are translated & untranslated so they work

Other System Resources
Diversity System Functional Architecture:
Payload Execution

Modified loader transforms original stored program and generates wrapper that retranslates external calls.

Response to normal user inputs are translated & untranslated so they work.

Other attacks fail because injected commands are wrong.
Conclusions

• Exploits are fragile
• Many techniques available to introduce synthetic diversity
• Randomization and “effective size of space” are key
  – Low run-time overhead
  – Load time, memory usage
• Diversity is not a panacea
• Next steps:
  – Currently building a synthetic diversity system (DAWSON)
  – Under DARPA sponsorship
Backup
Illustrative Run-Time Transforms

These transforms all make use of assumptions about the run-time representation used by programs.

– T1: Shift the position of the stack pointer on the runtime stack
– T2: Shuffle the jump table
– T3: Rename file system resources
– T4: Rename ports for local process
– T5: Put return pointers on separate stack of their own
– T6: Encrypt file name parameters to system calls
– T7: Perform simple obfuscation transforms on code
– T8: Break the run-time stack into multiple stacks, so the return pointer stack is far away from any buffers.
– T9: Optimize away stack references – inline return addresses
# Illustrative Source Code Transforms

<table>
<thead>
<tr>
<th>N-version programming</th>
<th>Or different compilers and hardware platforms</th>
</tr>
</thead>
</table>
| Diversity in functional behavior in components | Different encryption algorithms  
Different scales for data (e.g., Celsius or Fahrenheit) |
| Semantic-preserving source code transforms | Place sensitive data (e.g., function and data pointers) below starting address of any buffer  
- Reorder local variable to place buffer after pointer  
- Copy pointers in function arguments to area preceding local variable buffers  
Variable ordering  
Equivalent instructions |
| Variable compilation | Variable internal names  
Variable padding and addresses  
Variable linking order |
Taxonomies of Vulnerabilities & Attacks


• Richardson, T.W., *The Development of a Database Taxonomy of Vulnerabilities to Support the Study of Denial of Service Attacks*, PhD thesis, Iowa State University, 2001
Vulnerabilities and Exploits

- w00w00, “Heap Overflow”, http://www.w00w00.org/files/articles/heaptut.txt, 1/1999
Software Fault Tolerance & N-version Programming

Obfuscation -- Java Code

Obfuscation -- Protecting Software


- w00w00, “Heap Overflow”, http://www.w00w00.org/files/articles/heaptut.txt, 1/1999


- Gregory Wroblewski; “General Method of Program Code Obfuscation,” 2002 International Conference on Software Engineering Research and Practice (SERP’02), June 24 - 27, 2002, Monte Carlo Resort, Las Vegas, Nevada, USA


Source Code Transforms to Mitigate Vulnerabilities

- StackGuard, Libverify, RAD, PointGuard, MS C++ compiler
Run-time Transforms to Mitigate Vulnerabilities


Technical Approach

• Apply program transformation techniques to generate diversity
• Focus on remote attacks
• Specification based transformation
  – V-SPEC variant, A-SPEC invariant
• Apply to binary code, libraries & DLL’s
  – Don’t assume presence of source code or original compiler
  – Allow transforms may make use of characteristics of particular OS, or COTS products
• Build on recent work in area of code transformation and analysis
  – Leverage transformations developed for code obfuscation (Wang; Wroblewski;
  – Static analysis, program slicing (Weiser, Horowitz)
  – Cryptographic transformations
  – Software synthesis (Dijkstra, Gries, Schneider)
• Evaluate benefits of diversity thus achieved
  – theoretical and practical effectiveness (computational complexity measures, experiments)
  – with respect to different classes of attacks
  – with respect to programming flaws and errors
• Produce a prototype tool that can be used to introduce needed diversity into existing widely distributed and vulnerable COTS products though mutation and transformation.
  – Tool uses a user extensible library of transformations
  – Can alter programs via mutation on-the-fly