Review and Analysis of Synthetic Diversity for Breaking Monocultures

James E. Just

jjust@globalinfotek.com

Mark W. Cornwell

mcornwell@globalinfotek.com

WORM-04

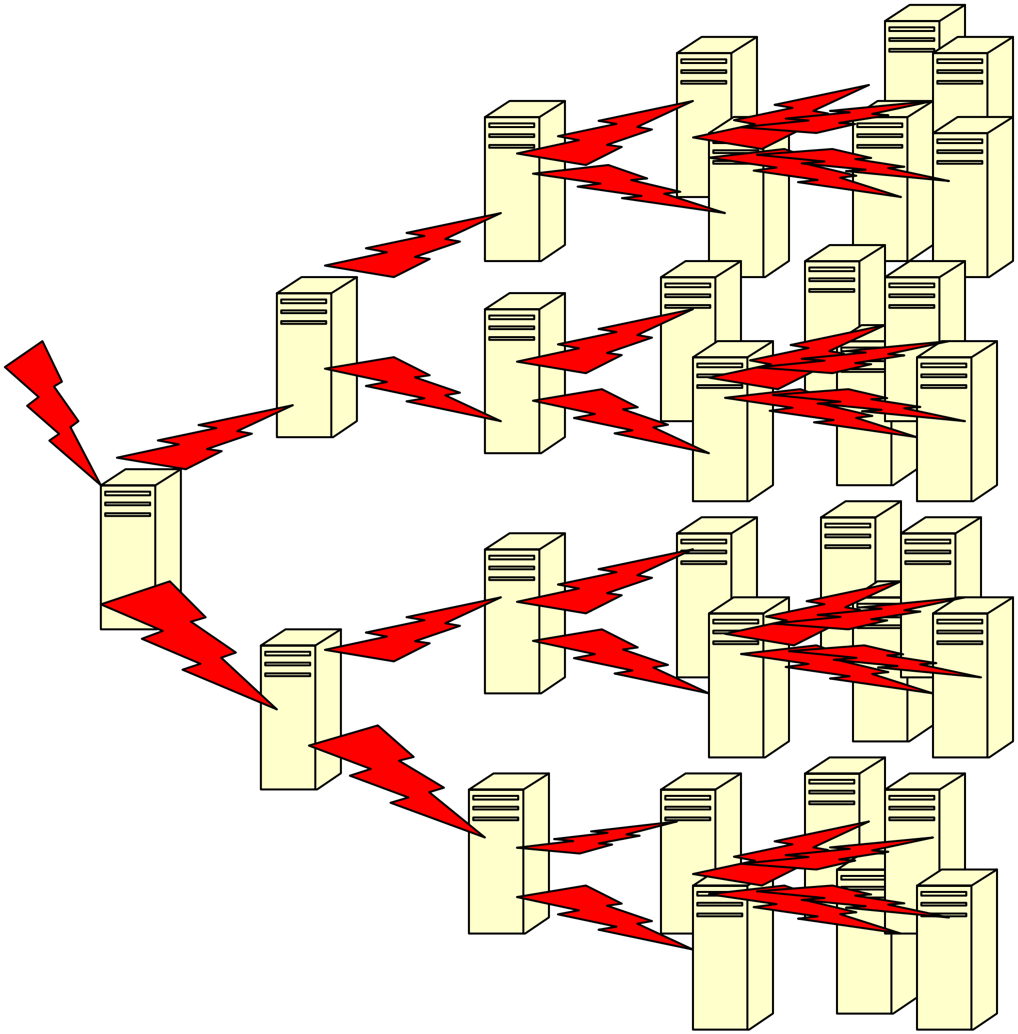
29 October 2004

Global InfoTek, Inc.

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



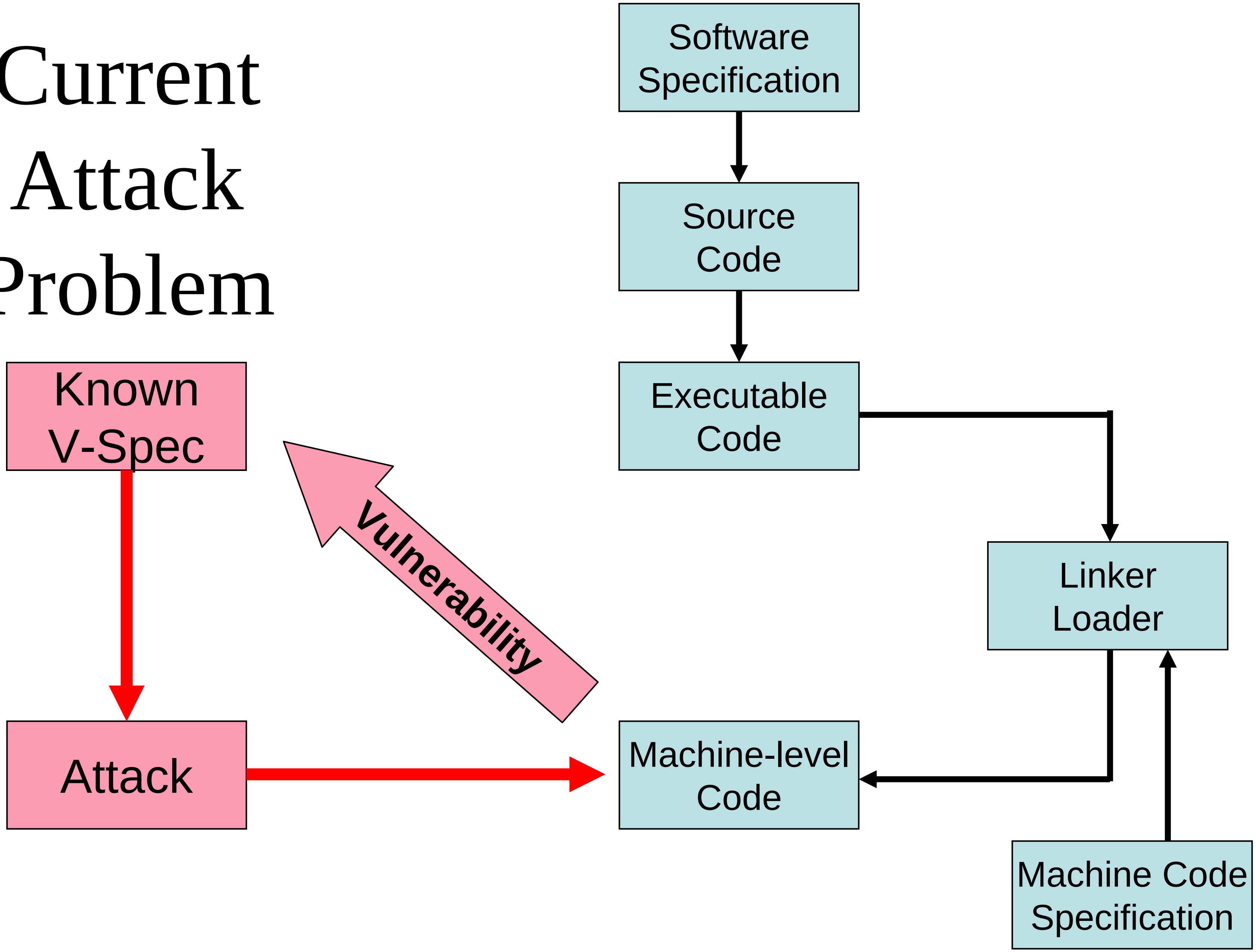
- Homogenously vulnerable host population
- Attack finds rich environment to exploit and spread
- Exponential growth leads to catastrophic failure

How to prevent exponentially cascading failures?

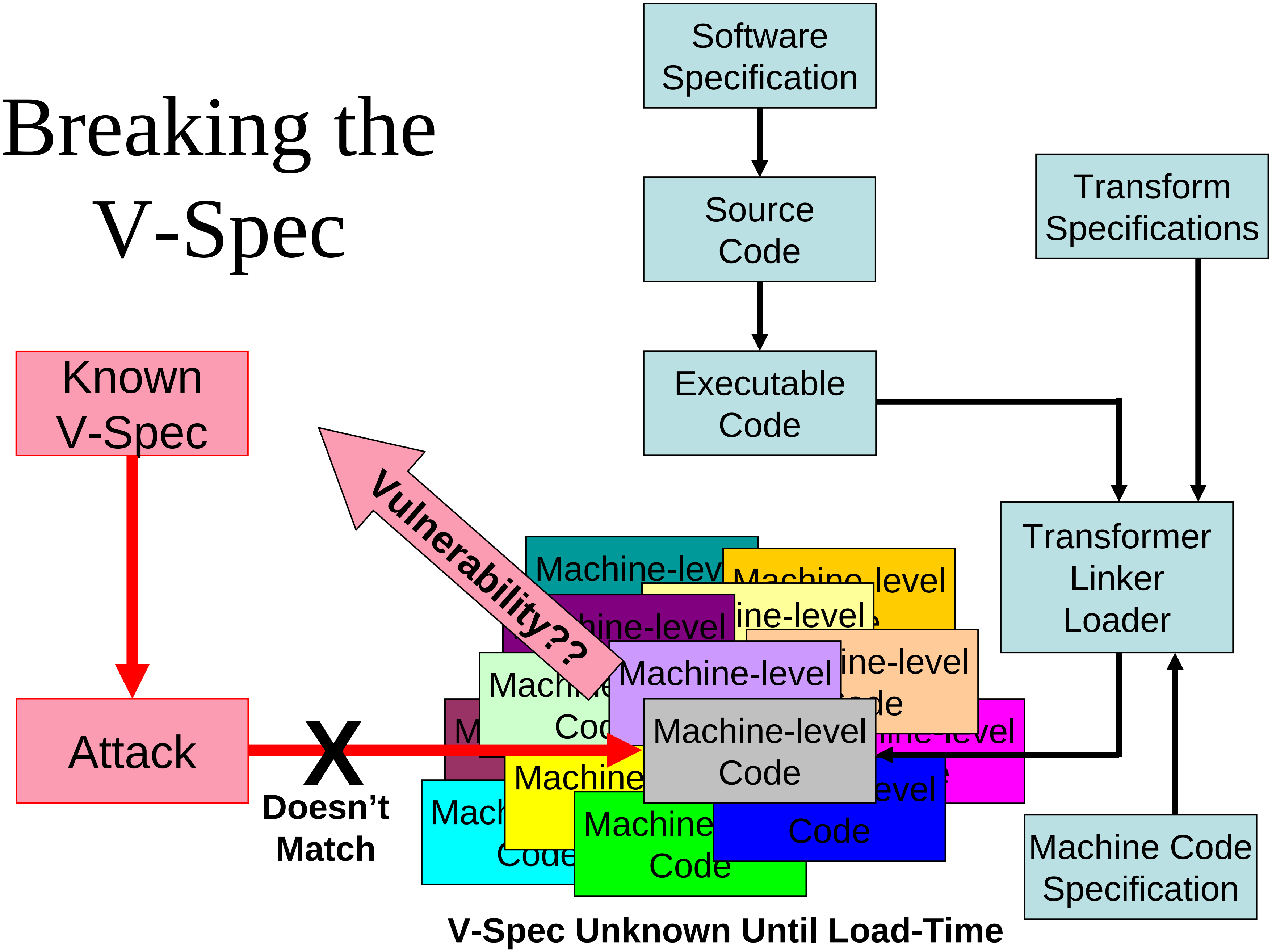
Problem Space (2): Difficulty of Building Intrusion Tolerant Systems

- Intrusion tolerant systems difficult to build
  - Expensive, large hw/sw footprints, *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



Breaking the V-Spec



Illustrative Common Techniques & Assumptions

Segment of Code Red 1 Disassembly[1]

```
loc_4B4:        ; CODE XREF: DO_RVA+26Dj
mov  esi, esp
mov  ecx, [ebp-198h]; set ecx with the data segmen
push  ecx; push data segment (pointer of f                to lo
mov  edx, [ebp-1CCh]; get current RVA base offset
push  edx; push module handle(base loaded address)
call  dword ptr [ebp-190h]; call GetProcAddress
```

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.DLLto access the file system, open network sockets and send and receive network packets.

s by Ryan Permeh, Marc Maiffret. http://www.eeye.com/html/advisories/codered.zip.

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 algortihm-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 aprogram, 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

Program	Combination (1)		Combination (2)	
	% Overhead	Standard Deviation (% of mean)	% Overhead	Standard Deviation (% of mean)
tar	-1	3.4	0	5.2
wu-ftpd	0	1.4	2	2.1
gv	0	6.1	2	2.1
bison	1	2.0	8	2.3
groff	-1	1.1	13	0.7
gzip	-1	1.9	14	2.5
gnuplot	0	0.9	21	1.0

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

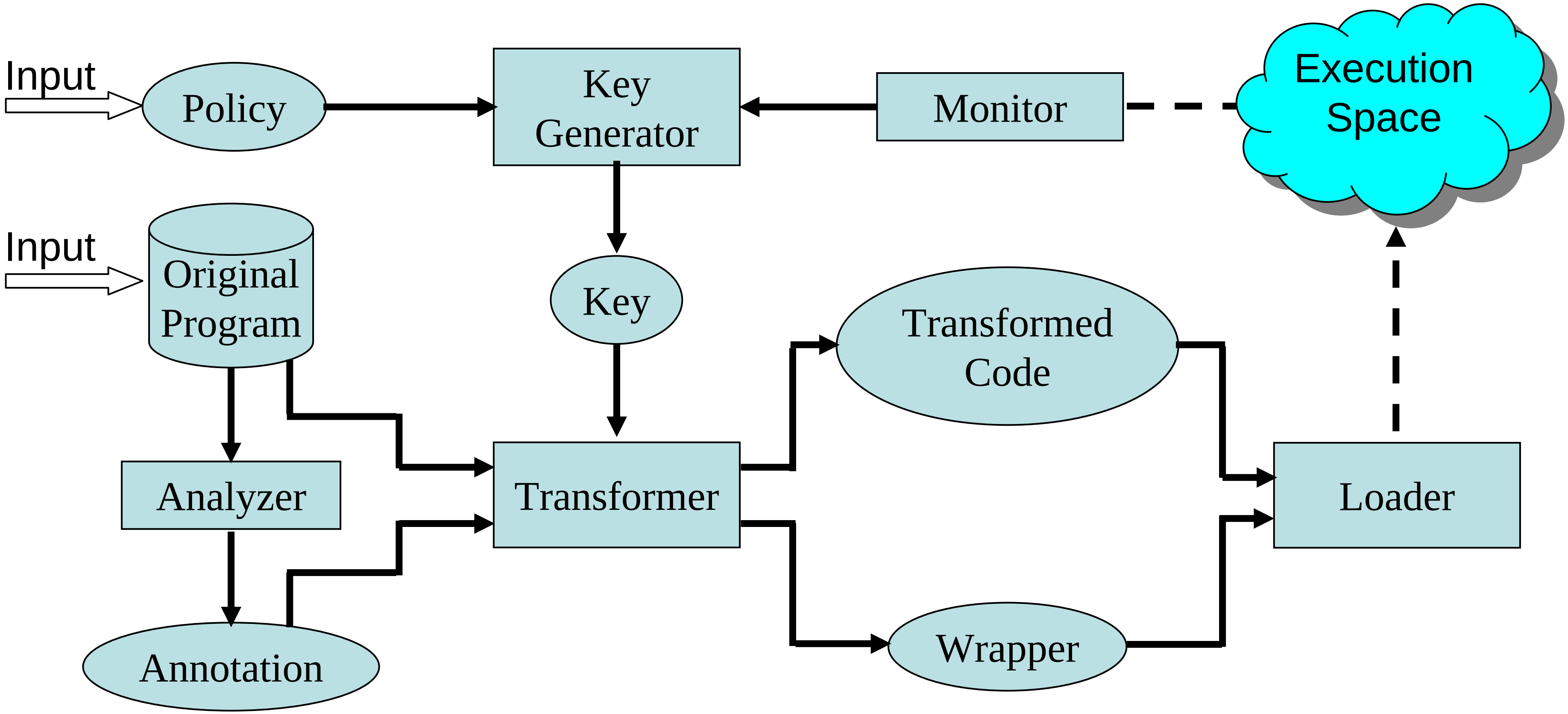
Effectiveness of Transforms

> Diversity is not a panacea for achieving cyber-security. There are many other ways to penetrate a system

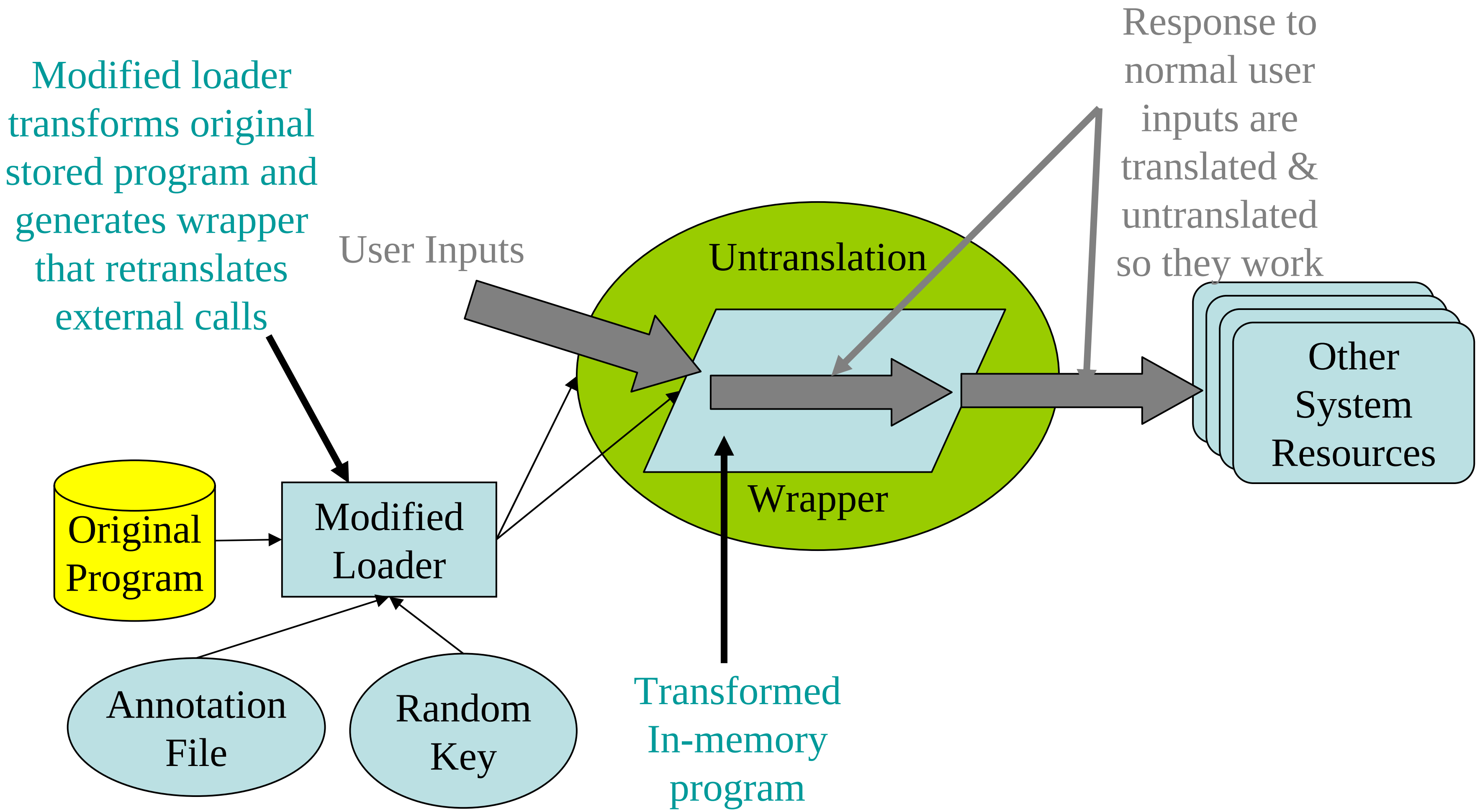
Vulnerability Class	Sample Vulne… s in Class	N-version Programming	Behavior Transforms (Source Code)	Internal Data Flow Transforms (Source Code)	Internal Control Flow Transforms (Source Code)	Place Sensitive Data Below Buffers (Source Code)	Encrypt Executable	Obfuscate Executable Code	Transform Relative Addresses (Executable)	Transform Absolute Addresses (Executable)	Transform System Resources (Executable)	Transform (Virtual) Host Instruction Set	Transform Protocols State Space (Source)	Transform Executable Protocols via Proxies	Encrypt Network/Interface Protocols
Improper Validation	**Memory error**														
	Code injection a… s (stack, heap, and integer ove… ws)	+	+	+	+	+	•	< [1]	++	++	++	++	•	•	•
	Existing code exploits (pr… f)	+	+	+	+	•	•	< [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	•	•	•	•	•	•	•	•	•	•	•	•	•	•
Inadequate Randomness	**Weak passwords**	•	•	•	•	•	•	•	•	•	•	•	•	•	•
	Weakly encrypted passwords	+	•	•	•	•	•	•	•	•	•	•	•	•	•
Improper Deallocation	**Memory leakage**	+	•	•	•	•	•	•	•	•	•	•	•	•	•
	Resource exhaustion	+	•	•	•	•	•	•	•	•	•	•	+	+	•

Key: Effectiveness in preventing attack: ++ = highly effective, + = effective, • = negligable, < = potentially counterproductive

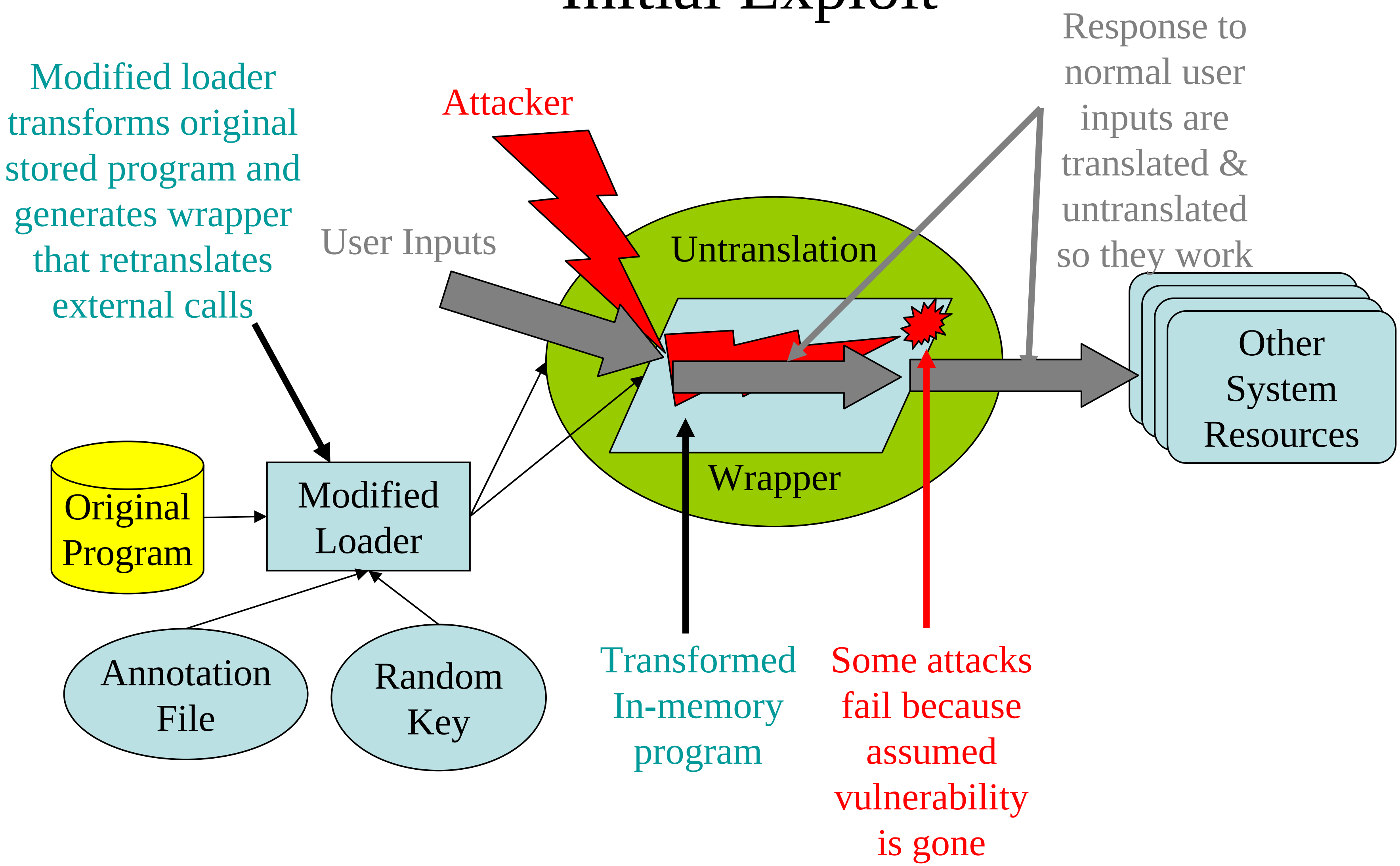
Diversity System Architecture Overview



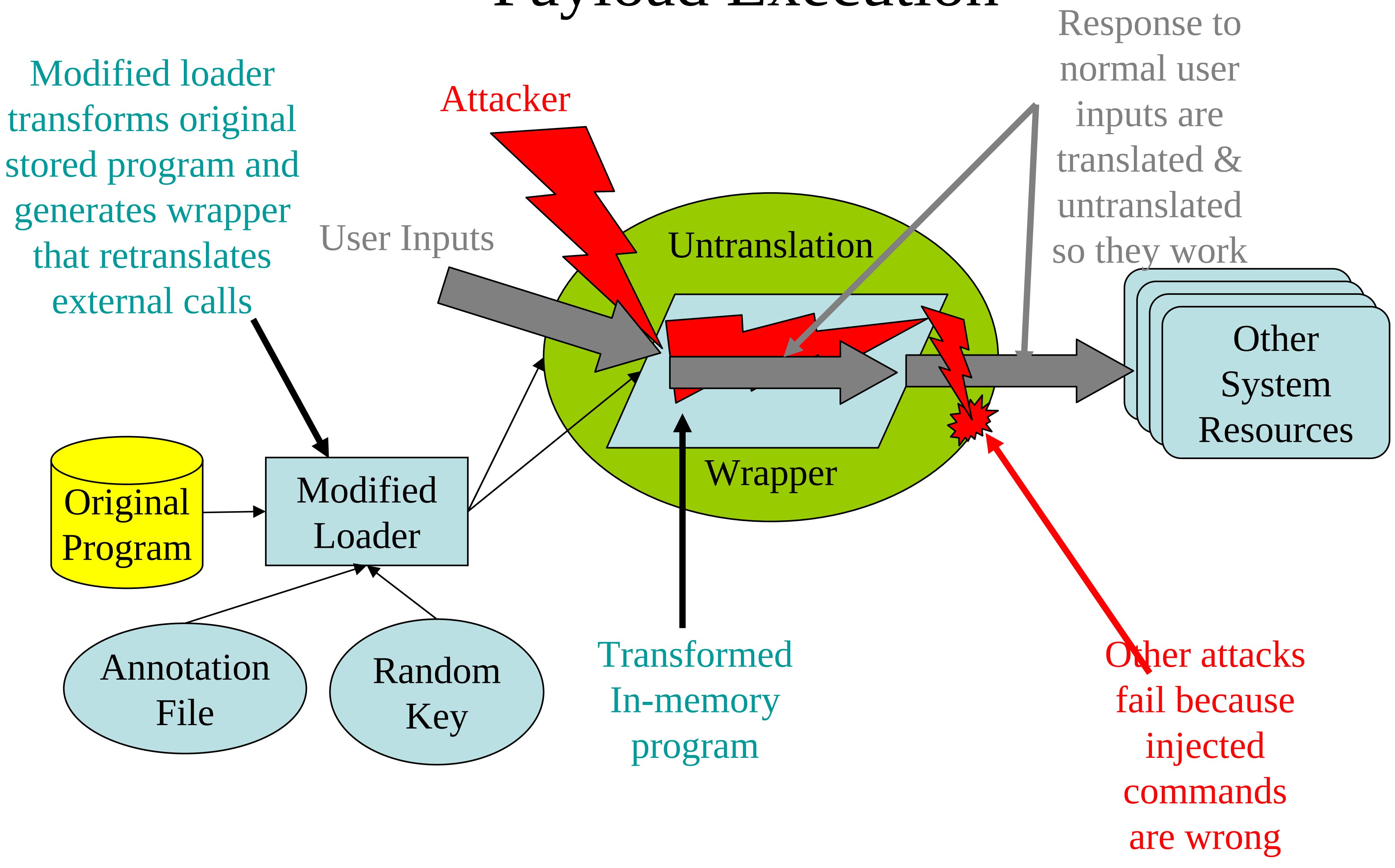
Diversity System Functional Architecture: Normal



Diversity System Functional Architecture: Initial Exploit



Diversity System Functional Architecture: Payload Execution



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

N-version programming	• Or different compilers and hardware platforms
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

- Aslam, T.. *A Taxonomy of Security Faults in the Unix Operating System*. Master's Thesis, Purdue University, Department of Computer Sciences, August 1995. http://citeseer.nj.nec.com/aslam95taxonomy.html
- Du, W. and Mathur, A, *Categorization of Software Error that Led to Security Breaches*, Technical Report 97-09, Purdue University, Department of Computer Science, 1997
- Krsul, I.V., *Software Vulnerability Analysis*, PhD thesis, Purdue University, West Lafayette, IN, May, 1998, p. 17, available at https://www.cerias.purdue.edu/techreports-ssl/public/97-05.pdf
- Landwehr, C. E., Bull, A. R., McDermott, J. P., Choi, W. S., "A Taxonomy of Computer Program Security Flaws." *ACM Computing Surveys*, Volume 26, Number 3, September 1994
- Lough, D.L., *A Taxonomy of Computer Attacks with Applications to Wireless Networks*, PhD Thesis, Virginia Polytechnic and State University, Blackburg, VA, available at http://scholar.lib.vt.edu/theses/available/etd-04252001-234145/
- 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

- Aleph One, “Smashing The Stack For Fun And Profit”, Phrack 49, Volume Seven, Issue Forty-Nine, File 14 of 16, 11/8/1995
- David Litchfield, “Defeating the Stack-Based Overflow Prevention Mechanism of Microsoft Windows 2003 Server”, NGS Research Whitepaper, August 9, 2003, http://www.nextgenss.com/papers.htm
- Mudge, “How To write buffer overflows”, http://www.insecure.org/stf/mudge_buffer_overflow_tutorial.html, 10/20/1995
- w00w00, “Heap Overflow”, http://www.w00w00.org/files/articles/heaptut.txt, 1/1999
- Ryan Permeh, Marc Maiffret, Code Red Disassembly Analysis, eEye Digital Security, http://www.eeye.com/html/advisories/codered.zip.
- Stuart Staniford, Nicholas Weaver, Vern Paxson. “Flash Worms: Is there any Hope?” Silicon Defense, Retrieved 27 March 2003 <http://silicondefense
- Stuart Staniford, Vern Paxson, Nicholas Weaver. “How to Own the Internet in Your Spare Time”, Proceedings of the 11th USENIX Security Symposium. August 2002, Retrieved 27 March 2003, <http://www-dirt.cs.unc.edu/netlunch/fall02/SPW02-worms.htm>

Software Fault Tolerance & N-version Programming

- A.Avizienis, “Fault Tolerance and fault intolerance. Complimentary approaches to reliable computing”, Proc. 1975 Int. Conf. Reliable Software, Los Angels, CA, Apr 21- 27, 1975, pp 458 - 464
- A.Avizienis, “N-Version Approach to fault tolerant Software”, IEEE-Software eg., vol- SE11, No12, Dec 1985, pp.1491 -1501
- V. Bharathi, “N-Version programming method of Software Fault Tolerance: A Critical Review”, Indian Institute of Technology, Kharagpur 721302, December 28-30, 2003
- L. Chen and A. Avizienis, "N-version programming: A fault-tolerance approach to reliability of software operation," IEEE 8th FTCS, pp. 3-9, 1978
- J.C. Knight and N.G. Leveson, “A Large Scale Experiment In N-Version Programming”, Digest of Papers FTCS-15: Fifteenth International Symposium on Fault-Tolerant Computing, June 1985, Ann Arbor, MI. pp. 135-139.
- J.C. Knight and N.G. Leveson, “An Experimental Evaluation of the Assumption of Independence in Multi-version Programming”, IEEE Transactions on Software Engineering, Vol. SE-12, No. 1 (January 1986), pp. 96-109.
- M.R. Lyu, J.-H. Chen, and A. Avizienis, "Software diversity metrics and measurements," In Proc. The Sixteen Annual Int. Computer Software and Applications Conf. 1992, pp. 69-78.

Obfuscation -- Java Code

- C. Collberg, C. Thomborson, and D. Low. “A Taxonomy of Obfuscating Transformations”. Technical Report 148, Department of Computer Science, University of Auckland, July 1997.
- C. Collberg, C. Thomborson, and D. Low. “Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs” Department of Computer Science, University of Auckland. ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'98). January 1998
- C. Collberg, C. Thomborson, D. Low. “Breaking Abstractions and Unstructuring Data Structures”, Proceedings of the 1998 International Conference on Computer Languages, pages 28-38. IEEE Computer Society Press. May 1998.
- Larry D’Anna, Brian Matt, Andrew Reisse, Tom Van Vleck, Steve Schwab, Patrick LeBlanc, “Self-Protecting Mobile Agents Obfuscation Report - Final report,” Network Associates Laboratories, Report #03-015, June 30, 2003
- Lee Badger, Larry D'Anna, Doug Kilpatrick, Brian Matt, Andrew Reisse, Tom Van Vleck. “Self-Protecting Mobile Agents Obfuscation Techniques Evaluation Report,” Network Associates Laboratories, Report #01-036, Nov 30, 2001, updated March 22, 2002.
- Douglas Low, Java Control Flow Obfuscation, MS Thesis, Univ. Auckland, 3 June 1998

Obfuscation -- Protecting Software

- Boaz Barak, Oded Goldreich, Russell Impagaliazzo, Steven Rudich, Amit Sahai, Salil Vadhan, and Ke Yang. “On the (im)possibility of obfuscating programs.” In J. Kilian, editor, Advances in Cryptology-CRYPTO ‘01, Lecture Notes in Computer Science. Springer-Verlag.
- Stanley Chow, Philip A. Eisen, Harold Johnson, Paul C. van Oorschot: A White-Box DES Implementation for DRM Applications. Digital Rights Management Workshop 2002: 1-15
- S. Chow, P. Eisen, H. Johnson and P.C. van Oorschot, ``White-Box Cryptography and an AES Implementation", Proceedings of the Ninth Workshop on Selected Areas in Cryptography (SAC 2002)
- Matthias Jacob, Dan Boneh, and Edward Felten. Attacking an obfuscated cipher by injecting faults , 2002 ACM Workshop on Digital Rights Management. Washington, D.C., 2002
- Hamilton E. Link and William D. Neumann, “Clarifying Obfuscation: Improving the Security of White-Box Encoding”, Sandia National Laboratories, Albuquerque, NM, downloaded from eprint.iacr.org/2004/025.pdf
- Chenxi Wang, “A Security Architecture for Survivability Mechanisms.” PhD thesis, University of Virginia, October 2000.
- Chenxi Wang, "Protection of software-based survivability schemes", in the proceedings of 2001 Dependable Systems and Networks. Gutenburg, Sweden. July 2001.
- w00w00, “Heap Overflow”, http://www.w00w00.org/files/articles/heaptut.txt, 1/1999
- Gregory Wroblewski, “General Method of Program Code Obfuscation,” PhD Dissertation, Wroclaw University of Technology, Institute of Engineering Cybernetics, 2002.
- 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

- Hamilton E. Link and William D. Neumann, “Clarifying Obfuscation: Improving the Security of White-Box Encoding”, Sandia National Laboratories, Albuquerque, NM, downloaded from eprint.iacr.org/2004/025.pdf

- Cullen Linn, Saumya Debray, “Obfuscation of Executable Code to Improve Resistance to Static Disassembly,” ACM Conference on Computer and Communications Security, Washington DC, October 27-31, 2003.

Source Code Transforms to Mitigate Vulnerabilities

- M. Chew, D. Song. “Mitigating Buffer Overflows by Operating System Randomization,” Technical Report CMU-CS-02-197.
- Hiroaki Etoh and Kunikazu Yoda. Protecting from stack smashing attacks. Published on World-WideWeb at URL http://www.trl.ibm.com/projects/security/ssp/main.html, June 2000.
- Stephanie Forrest, Anil Somayaji, and David H. Ackley. “Building diverse computer systems.” In 6th Workshop on Hot Topics in Operating Systems, pages 67-72, Los Alamitos, CA, 1997. IEEE Computer Society Press.
- Selvin George, David Evens, Steven Marchette. “A Biological Programming Model for Self-Healing”, First ACM Workshop on Survivable and Self-Regenerative Systems (in association with 10th ACM Conference on Computer and Communications Security) October 31, 2003, George W. Johnson Center, George Mason University, Fairfax, VA
- Pax. Published on World-Wide Web at URL http://pageexec.virtualave.net, 2001.
- Jun Xu, Z. Kalbarczyk and R. K. Iyer. “Transparent Runtime Randomization for Security”. Proc. of 22nd Symposium on Reliable and Distributed Systems (SRDS), Florence, Italy, October 6-8, 2003
- StackGuard, Libverify, RAD, PointGuard, MS C++ compiler
- Peter Silberman and Richard Johnson, A Comparison of Buffer Overflow Prevention Implementations and Weaknesses, I-Defense, 1875 Campus Commons Dr. Suite 210 Reston, VA 20191, http://www.blackhat.com/presentations/bh-usa-04/bh-us-04-silberman/bh-us-04-silberman-paper.pdf

Run-time Transforms to Mitigate Vulnerabilities

- Elena Gabriela Barrantes, David H. Ackley, Stephanie Forrest, Trek S. Palmer, Darko Stefanovic and Dino Dai Zovi, “Randomized instruction set emulation to disrupt binary code injection attacks,” 10th ACM Conference on Computer and Communications Security, Washington DC, October 27-31, 2003.
- Sandeep Bhatkar, Daniel C. DuVarney, and R. Sekar, “Address Obfuscation: An Efficient Approach to Combat a Broad Range of Memory Error Exploits,” 12th USENIX Security Symposium, August 2003.
- Gaurav S. Kc, Angelos D. Keromytis, Vassilis Prevelakis, “Countering Code-Injection Attacks with Instruction-Set Randomization,” 10th ACM Conference on Computer and Communications Security, Washington DC, October 27-31, 2003.

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