Review and Analysis of Synthetic Diversity for Breaking Monocultures

James E. Just
Global InfoTek, Inc.
1920 Association Drive
Reston, VA 20191 USA
1-703-652-1600, x315
jjust@globalinfotek.com

Mark Cornwell
Global InfoTek, Inc.
1920 Association Drive
Reston, VA 20191 USA
1-703-652-1600, x234
mcornwell@globalinfotek.com

ABSTRACT

The increasing monoculture in operating systems and key applications and the enormous expense of N-version programming for custom applications mean that lack of diversity is a fundamental barrier to achieving survivability even for high value systems that can afford hot spares. This monoculture makes flash worms possible. Our analysis of vulnerabilities and exploits identifies key assumptions required to develop successful attacks. We review the literature on synthetic diversity techniques, focusing primarily on those that can be implemented at the executable code level, since this is where we believe there is the most potential to reduce the common mode failure problem in COTS applications. Finally we propose a functional architecture for synthetic diversity at the executable code level that reduces the common mode failure problem in COTS applications by several orders of magnitude.

Categories and Subject Descriptors

D.1.0 Software: Programming Techniques, General

General Terms

Performance, Design, Security,

Keywords

Diversity, Vulnerability, N-version programming

1. INTRODUCTION

There is a great desire for affordable, robust systems that respond automatically to accidental and deliberate faults. The current state of the art employs fault-tolerance technologies for accidental faults and errors and intrusion-tolerance technologies for malicious, intentional faults caused by an intelligent adversary. Combining fault- and intrusion-tolerance technologies can produce very robust and survivable systems.

However, such systems have an Achilles heel. Their robust performance depends upon the continued existence of spare resources for failover. Spare resources can be depleted by a determined adversary simply by continued attacks until the system can no longer maintain critical functionality. The dearth of alternative COTS operating systems, applications and hardware platforms and the expense and questionable effects of N-version programming [2], [3][2], [10], [24], [25], [31] for custom applications mean that lack of diversity is a fundamental barrier to achieving true robustness, even for high value systems that can afford hot spares.

The paper is motivated by our belief that fine-grained synthetic diversity is possible and that such techniques can be used to break the implied software vulnerability specification that an attacker depends on for successful exploits without breaking the required functionality of the component. We focus on generating diversity at the executable code level, which is where we believe the greatest leverage exists to defeat attacks on COTS application. We draw on our previous experience in building intrusion tolerant systems [22]. When deployed widely, the mechanisms described in the paper, if successful and practicable, should introduce enough spatial and temporal diversity into the current world-wide computer monoculture to significantly reduce, if not eliminate, the ability of attackers to “take over the Internet in their spare time” [39]. Such mechanisms would also enable the generation of essentially unlimited spares for robust, intrusion tolerant systems.

While local diversification can protect single hosts from many attacks, some attacks exploit weaknesses in the network protocols enable services across networked hosts. There are novel techniques for introducing heterogeneity into common network protocols in ways that will thwart an outside attacker but leave the normal network operations of the system unchanged. The basic ideas of these transformations are to add new states to the state machine specification of correct protocol behavior to modify either message content or sequencing or to facilitate monitoring. Unfortunately, the complexity of the topic precludes further discussion in this paper. Techniques for protocol diversity will be the subject of another paper. Diversity at the level of interpreting scripting languages can be effective in analogous ways [23].

It must be noted that diversity is not a panacea for all cyber-attack problems. It cannot mitigate many vulnerabilities such weak passwords, default accounts, cross site scripting, and denial of service.

2. BREAKING VULNERABILITY SPECIFICATIONS -- OUR APPROACH TO TRANSFORMS FOR DIVERSITY

Binary code and network protocols share the characteristic that they are the integration points for most software. The source code for these systems is compiled and then integrated at the binary level for a specific operating system (Windows/Solaris/ Linux) on particular hardware architecture (Intel/Sparc). What binaries are to individual hosts, protocols (e.g., TCP/IP) are to integrating multiple hosts. They provide common interfaces (abstract machines) that interpret code according to highly specific rules and conventions.

WORM’04, ***OCTOBER 29, 2004, WASHINGTON, DC, USA.***

Our approach destroys both the ability of attackers to inject code that executes effectively and their ability to exploit existing code to do their bidding. We do so by altering the interface and representation conventions in such a way that injected code no longer functions and existing code is no longer reachable. For binary code, such conventions include managing the run-time environment, calling library routines, and addressing in-memory tables. We randomize critical information that is normally assumed to be static and predictable by attacking code. [1]

Program transformation can be described in the framework of software verification. In this framework, we view specifications as objects that capture assumptions about software. Specifications are more abstract than the programs themselves, because many different programs may implement the same specification. A tenet of software engineering is that anyone writing a program that uses another program only uses the assumptions that are allowed by the specification. One who has obeyed this tenet enjoys the benefit that his program will work with any other program that implements the same specification. Those who disobey are faced with programs that may break when other programs are substituted.

The author of any attack, such as Code Red, must identify both the specific vulnerability details in a widely deployed application and how to exploit that vulnerability to start the attack. Details are important because most attacks are executable code and every bit counts. For example, the author might identify a flaw such as a buffer overrun that allows one to write data into the runtime stack or heap. He must then identify specific locations that are branching addresses and exploit them to point to his injected code. This injected code must find and execute system calls at the binary level to access system resources, talk over the network, propagate itself further, and do whatever other tasks the author intends.

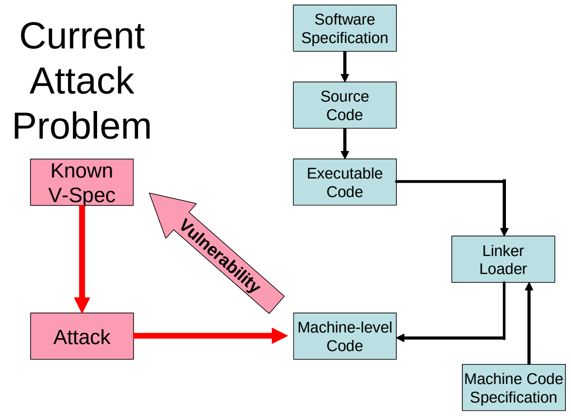

Figure 1. Assumptions Made by Attackers

These details can be viewed as a vulnerability specification from the perspective of the attacker. A program that supports the assumptions of the attacker supports the Vulnerability Specification or V-SPEC and is vulnerable to that attack. This is illustrated in Figure 1 above.

Programs not supporting the V-SPEC are invulnerable to that particular attack. Our approach transforms programs in ways that break the V-SPEC, without affecting legitimate assumptions about program behavior, which we refer to as the A-SPEC. It is not necessary that we know the precise A-SPEC in order to make use of the concept. We merely need confidence that, after applying transformations, the runtime executable still conforms to the A-SPEC. For example, if random sized blocks of information are pushed onto the stack to make stack locations harder to predict, legitimate programs should not be affected. Such assumptions underlie much of software obfuscation. In our terminology, ideal diversity inducing transformations will have the property that they are both V-SPEC variant and A-SPEC invariant. This concept is illustrated below in Figure 2.

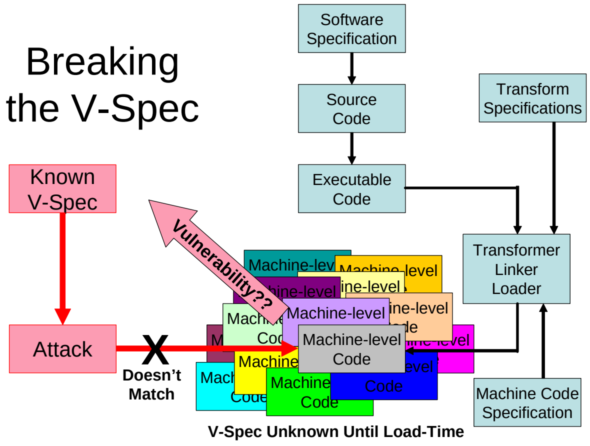

Figure 2. Invalidating Attacker's Assumptions

We believe that transforms can be used to both make widespread exploitation of common mode vulnerabilities more difficult but also to make it more difficult for an attack to propagate attacks if an exploit succeeds.

3. RELATED WORK ON TRANSFORMATIONS

There exists a rich body of research and practical experience on the topic of program transformations that we can leverage. Application of transformation techniques to make code more resistant to code injection attacks is a relatively recent area [19]. We begin with a review of the transforms used for obfuscation since they have been around the longest.

3.1. Obfuscation to Preserve Secrets

Code obfuscation refers to techniques that can be applied to software to make reverse engineering of code more expensive in terms of time and effort while preserving its functionality. It has traditionally been used as a method of protecting intellectual property embodied in software. Interest in the area soared as Java became popular because decompiling Java code was easy.

Collberg, Thomborson, and Low at the University of Auckland published the first systematic study of Java code obfuscation, a taxonomy of obfuscation techniques [14].Its categories included layout transformations (e.g., scrambling identifiers, removing comments, changing formats), control flow transformations (e.g.,

[1] We believe that it is possible to implement a similar scheme for network protocols, e.g., changing the state space of the protocol and the encodings of messages by which the protocol interacts. This topic will be the subject of a separate paper.

statement grouping, ordering, computation, and opaque constructs), data transformations (storage, encoding, grouping, ordering) or preventative transformations (e.g., exploiting weaknesses of decompilers).

They also produced algorithms for low-cost, stealthy, opaque constructs useful for obscuring program control flow [15] and developed techniques for obscuring data structures and abstractions like classes and built-in data types [16]. They measured effectiveness of obfuscation using various software engineering complexity metrics. Low implemented a complex obfuscating algorithm in his Masters thesis [30].

Wang [40] studied the malicious host problem, specifically how to protect trusted probe software (that contained no exploitable faults and whose design mandated frequent communication with a remote trusted host controller) so that it maintained algorithm privacy and execution integrity. Both denial of service attacks and random tampering with the executing probe were ignored. She argued that probe impersonation, intelligent tampering with the probe, and spoofing the inputs to the probe were the most significant threats.

While input spoofing in general is an unsolvable problem, Wang stated [ibid, p. 8], "if spoofing input x requires solving the algorithm-secrecy or execution-integrity problem, then techniques to ensure the latter can be used to counteract input spoofing. However, there are applications where this is not possible." Her premises hold for calculations that involve clock time on the potentially malicious host but not for calculations that involve values known only to the local host. As part of her proof of concept, she proves that, because of the pervasive aliasing, precise analysis of the transformed program (recovery of the CFG) is NP hard and provides some empirical verification of the results using both brute force search and alias approximation. Her performance analysis of the resulting transformed code found that replacing 50% of the branches in typical program results in an increase in execution time of a factor of 4 and almost a doubling in its size.

Diversity for intrusion tolerance is similar to some aspects of the malicious host problem that Wang studied; however, it differs in a number of fundamental ways. First, vulnerabilities in the protected program cannot be ignored or assumed away. For example, protecting carefully-crafted software probes that are constantly monitored by remote secure hosts is very different than building systems from COTS components that are smart enough to protect themselves and learn from their mistakes. Second, with COTS programs, access to program source code is not guaranteed so protection must be achieved at the binary executable level. Most importantly, our criteria for success are different. We are not trying to prevent reverse engineering. Our goal is to break an attacker's code. Since our focus in on executable code, most of the Collberg and Wang techniques are not applicable. They are included because of the strength of her theoretical results.

Wroblewski [41] generalized the Collberg et al. and Wang work by developing and implementing a purely sequential algorithm that provided controllable obscuration (via a rescaling factor that controls the growth in the size of the program) and that worked on binary code rather than source code. He proved that allowable transformations on executable code must meet one of the three conditions. They must be (1) reversible operations, (2) substitutions of equivalent instructions, or (3) any operations that change context not used by the original program (i.e., that portion of the computer's state that is not changed by the original program). He argued that only empirical research can answer the question of how well an algorithm protects a program from tampering and found, for example, that automatic creation of opaque constructs is weak because such programs generate similar patterns and are easily found and removed by humans even if they cannot be found and removed automatically. His actual empirical studies were very small scale.

Linn and Debray [27] approached obfuscation by disrupting the two major static disassembly approaches for executable code analysis (linear sweep and recursive traversal). They do not deal with dynamic disassembly techniques. Their implementation, based on a binary rewriting system developed for Intel IA-86 executables called PLTO, currently performs junk insertion and transformation of unconditional jumps and call instructions. Other transformations are being worked on. In testing, the best commercially available disassembly tool failed on 65% of the instructions and 85% of the functions. Their techniques resulted in about a 13% increase in execution time and 15-20% growth in size of the executable.

Barak's seminal proof [6], published shortly after Wang's study, showed that it is impossible to completely obscure code and that no general obfuscator is possible. His results are very strong. Related to this, Badger et al. [4] began extending Wang's work to protect groups of mobile agents against malicious hosts. They later abandoned the effort because they were unable to prove that any obfuscation method would resist reverse engineering efforts for any minimum time and subsequently refocused their study on better understanding obfuscation. Their resulting report [17]. is a *tour de force* on obfuscation theory and techniques and concluded that obfuscation should not be depended on for security, that Barak's results are very strong (general obfuscation is impossible and only specific secrets can be protected), and that, in many use cases, better solutions than obfuscation are available if the formulation of the problem is restated somewhat.

Obfuscation is a very important topic in the Digital Rights Management (DRM) since the class of problems that it must deal with involves protecting secrets from a malicious host, i.e., a host in which the adversary has complete control of execution and is able to run, stop, and restart the software at any point, reverse engineer components of the system, and see and manipulate all data. Chow, et al. [12] introduced the concept of white-box attack context to distinguish this problem from the traditional black-box threat context that is usually dealt with in cryptographic circles. They argued that, notwithstanding the Barak proof, their general approach "offers useful levels of security in the form of additional protection suitable in the commercial world, forcing an attacker to expend additional effort (compared to conventional black-box implementations)" [ibid, p. 2]. They implemented obscured DES and AES algorithms [12], [13]. Jacobs et al [21] demonstrated a differential fault injection attack on an obscured DES implementation. They argued that general problem of retrieving embedded data from a circuit is NP-hard so "no efficient general deobfuscator exists for this problem" [ibid, p. 10]. However, they admitted that this theoretical proof was not of much relevance since, even if the problem is hard in the worst case, it may be easy in most practical instances. Link and Neumann [26] improved on the Chow results above with an implementation which they claimed is secure against both the Jacobs et al. attack and the statistical bucketing attack technique that Chow describes [12], as well as a new adaptation of the statistical bucketing attack that they describe.

For our purposes, the above obfuscation techniques are only useful if they are essential to prevent attackers from easily determining our transformations so they can modify their attack dynamically. We believe that randomizing the choice of and specific implementation of the transformation techniques described below will provide a better solution to the monoculture problem. Our proposed solution will deliver spatial and temporal diversity that should thwart such attacks, i.e., .the implementation of run-time randomization of memory layout for COTS applications means that few users of a given application at a given time will have the same memory layout (hence spatial diversity) and that any given user over time will have different memory layouts (hence temporal diversity).

3.2. Mitigating Vulnerabilities in Source and Executable Code

Practical and theoretic claims of effectiveness against de-obfuscation attacks do not directly imply effectiveness against malicious code injection attack in our more general problem. It does not matter whether the code is intelligible to a human reader as long as the attacker can discover an exploitable vulnerability experimentally. Our goal is to invalidate whatever assumptions must be made in order for an attacker's malicious code to work. General approaches to mitigating problems in source code are listed below.

Table 1. Mitigating Vulnerabilities in Source Code

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

Wang [40], [41], Etoh and Yoda [18] and other authors have suggested the approaches listed above to mitigate vulnerabilities in source code. Note that these approaches do not identify or remove vulnerabilities in the source code but only transform existing source code so that it is less vulnerable or more diverse.

Forrest et al. [19] suggested the use of randomized transformation to introduce diversity into applications and prototyped the randomization of stack resident data addresses. They focused on buffer overflows (which are particularly widespread exploits, especially stack smashing attacks, and which predominately affect C and C++ programs) as an example. Their implementation modified the gcc compiler to insert random amounts of padding into each stack frame. Because buffer overflows are such a common problem, a variety of commercial solutions have been introduced such as StackGuard, Libverify, RAD, and PointGuard. Even the Microsoft C++ compiler has a setting that will prevent such overflows. The problem is getting developers to use them, particularly for COTS Windows applications where performance is a significant requirement.

Recently, other researchers have attempted to develop randomization as a practical approach to defeat buffer overflow and related attacks. Chew and Song [11] randomized the base address of the stack, system call numbers, and library entry points, through a combination of the Linux program loader modifications, kernel system call table modifications, and binary rewriting. Xu et al. [45] modified the Linux kernel to randomize the base address of stack, heap, dynamically loaded libraries, and GOT. The PaX project [33] modified the Linux kernel to randomize the base address of each program region: heap, code, stack, and data. Their approach remains vulnerable to attacks that rely on relative addresses between variables or code and to attacks that can access the base addresses of different memory segments but there are techniques available to mitigate these weaknesses, for example [18].

Forrest's group showed that scrambling an executable as it is loaded with a pseudorandom number seeded with a random key and then unscrambling it through a modified code emulator (implemented for the Intel x86 platform) was very effective in stopping injected code attacks from the network [7]. They paid a surprising low 5% performance penalty during testing, on a Pentium 200 MHz with 128 MB of memory, but increased memory usage by about a factor of three. They conclude that such a penalty may be acceptable on modern servers for the increase in security. They also discuss the dangers of randomly generating a valid instruction while descrambling injected code with the dense instruction set of the x86 chip. Their small test on this topic did not indicate any significant problems but they caution about the size of the test. Kc [23] produced similar results using a common randomization technique for all processes on the system.

Bhatkar et al [9] analyzed memory error exploits (e.g., buffer and integer overflows, return into libc) and characterized such attacks as either absolute address-dependent (such as overwriting code or data pointer) or relative address-dependent (such as overwriting non-pointer data). Their address obfuscation approach combated both types of memory error exploits by randomizing both the absolute locations of data and code and the relative distances between data locations. In their implementation, Bhatkar et al. randomized the base address of various memory regions (stack, heap, DLL, routines and static data) and permuted the order of variables and routines (e.g., local variables in stack frames, static variables, routines in shared libraries or in executables).Their implementation was limited to Linux but generally required no changes to the OS kernel or compilers and could be applied to individual applications rather than a whole system It transformed object files and executables at link-time and load-time; and had low run-time overhead (especially low for link-time transformation).

The protection offered by their technique against classic attacks such as stack smashing, existing code (return-into-libc), format string (e.g., printf), data modification, heap overflow and double-free, and integer overflow and the results was very encouraging.

However, their protection against data modification attacks if the data resides in the same stack or heap as the overflow was limited to non-existent. They suggested using the techniques described by Etoh and Yoda (see above) to help with this problem. The introduction of additional randomization in address obfuscation (e.g., random-sized gaps within stack frames and blocks allocated by malloc, reordering of and random padding within code and static variables) can also address these weaknesses. They next investigated three specifically crafted attacks that can defeat address obfuscation if the victim program contains the right vulnerability, particularly a bug that allows an attacker to read memory contents. Such a bug allows the attacker to succeed deterministically.

They point out [ibid], "the 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."

Bhatkar et al. collected performance impact data on a older Pentium III running the Linux operating system. The tested transforms included relocating the stack, heap, and code regions as well as the introduction of random gaps between stack frames. Their results for two different combinations are summarized in the table below.

Table 2. Performance Impacts of Selected 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 is static relocation performed at link-time. Combination 2 is dynamic relocation performed at load-time. They observed [ibid], "The overheads compared to conventional obfuscation transforms are quite modest. Combination 2 has noticeably more overhead because it requires position-independent code. However, when code is already being distributed in DLL form, combination (2) provides broad protection against memory error exploits without any additional overhead."

Responses to various protection approaches to stack smashing attacks and other exploit types are never far behind. Ever since Aleph One [1], Mudge [32], and w00w00 [42] popularized buffer overflows, the race for cures and counter-measures to them has been on. For example, Silberman and Johnson compare the effectiveness and work-arounds for seven different buffer overflow prevention techniques and four others that are explicitly incorporated into those seven [37]. The trend continues as David Litchfield described how to overcome the buffer overflow prevention scheme in Microsoft's Windows 2003 Server Operating System [28].

In summary, different approaches to mitigating vulnerabilities at the executable code level include instruction set transformations or randomization, transformations of system resource names, system call names or DLL names as well as various address transformations. We expect to implement most of these techniques except for instruction set transformations which are effective but degrade performance more than other equally effective techniques. If chipsets natively designed for variable instruction sets (like TransMeta' Crusoe chip) become popular, instruction set randomization may be much more practical.

4. TRANSFORMATIONS EFFECT ON VULNERABILITIES

Proof-of-concept systems, our own research efforts, and the literature provide measures of the effectiveness of various transformations against particular kinds of attacks. We have attempted to summarize, in Table 1 on the next page, the gist of these effectiveness measures at the transformation types level relative to selected specific attack types from the vulnerability classes in Lough's taxonomy [29]. It is difficult to be precise and comprehensive in such a table because of the enormous breadth of possible vulnerabilities and attacks. We have attempted to select an interesting subset of all the possible attack types within a given vulnerability characteristic. This is not to say that other attack types are not significant or that there are not variants of the attacks or variant of the transforms that might change the effectiveness of the approach. We offer the following observations on Table 1.

1. Memory errors are among the most fragile vulnerabilities with respect to program transformation. Many transformations applied with sufficient randomness to a population of hosts should diversify this class of vulnerability in a subject population.

2. Different transformations have different costs. N-version programming is expensive in terms of labor and can practically generate only a few versions of a program. Many obfuscating transforms incur a time and space penalty by inserting irrelevant code and computing dynamically quantities that could be represented statically in order to resist code slicing and reverse engineering attacks.

3. Different transformations are effective against different vulnerabilities. While some vulnerabilities are fragile and break under many transformations, others are unaffected by any transformations. As expected, transformations were not especially effective against exposure, randomness or deallocation errors. They were totally ineffective against weak passwords and dumpster diving.

4. Protocol transformations were effective against attacks that other transformations did not handle well. In particular protocol transformations appear effective against protocol errors that transformations deemed effective against memory errors didn't address.

5. Some obfuscation techniques may actually increase effectiveness of memory error attacks through a proliferation of pointers and computed branch addresses. This could produce a target rich environment for a memory error exploit that was generalized enough, especially if many computed addresses branch to positions close to one another.

Table 3. Transformation Effectiveness vs. Vulnerability Classes

Vulnerability Class	Sample Vulnerabilities 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 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	•	•	•	•	•	•	•	•	•	•	•	•	•	•
Inadequate Randomness	**Weak passwords**	•	•	•	•	•	•	•	•	•	•	•	•	•	•
	Weakly encrypted passwords	+	•	•	•	•	•	•	•	•	•	•	•	•	•
Improper Deallocation	**Memory leakage**	+	•	•	•	•	•	•	•	•	•	•	•	•	•
	Dumpster diving	•	•	•	•	•	•	•	•	•	•	•	•	•	•
	Resource exhaustion	+	•	•	•	•	•	•	•	•	•	•	+	+	•

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

5. FUNCTIONAL ARCHITECTURE FOR DIVERSITY

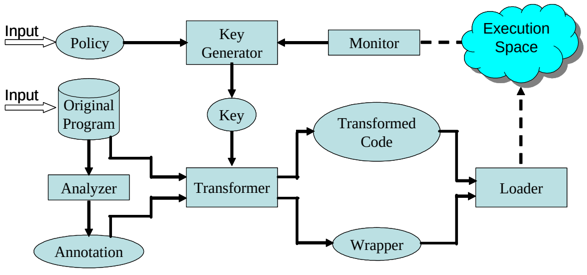

Figure 3. Functional Architecture for Diversity

A functional architecture for a system that would implement our diversity automation concept is shown in Figure 3 above.

The analyzer is least intuitive of the components in the diagram. There are some transforms that it is easy to do on executable code with just the loader. Others take more or less analysis at the binary or disassembled level. Still others can only be accomplished if the source compiler or a human has provided hints or annotations about the executable. The binary code analyzer in the diagram is an offline process. The idea is that the analyzer preprocesses the original executables and generates annotations from the analytic results that are useful to the transformer module. For example, it interprets linkage information in PE formatted executables and DLLs in order to locate entry points, unresolved absolute addresses, system call linkages, and help distinguish instructions from inline data values. It disassembles the code into an internal representation and builds control flow graphs (CFG), and data flow graphs (DFG), symbol maps, representational invariants, and other analysis structures that constitute higher level abstractions of the code's semantics. It then identifies and captures key information to enable the transformer to revamp both the relative and absolute memory layout of the executable without breaking its inherent functionality. A human or modified compiler could generate similar annotation files for an executable.

The transformer modifies the original program based on policy guidance and a random key. It uses meta-information provided by the annotation file to perform "smart" transformations (in the sense that they use the higher level abstractions from the annotations). For example, the transformer could shuffle the addresses of executable code by re-ordering basic blocks of code based on a map of the boundaries of those blocks provided by the CFG annotations. Any implementation of the transformer itself should be semi-interpretive, i.e., it should include a set of built-in, hard-coded transformation primitives. The primitives would be selected and combined randomly via an interpretive language that would generate a random memory layout of applications on each program load.

The output of the transformer includes transformed code and any wrappers (i.e., mediated connectors that intercept and modify DLL calls [5]) that the transformed code may need to communicate with its outside environment. Some simple operations such as offsetting the base of the run-time stack can be handled by prepending code to the executable to initialize stack pointers appropriately [9]. Fancier transformations such as changing the order of parameters to library routines may require a wrapper that supports a library routine with the changed parameter ordering.

The loader takes the transformed code and any wrappers it needs and loads them into the execution space. The loader component can be the standard loader so long as the transformed code and wrappers conform to the PE and DLL formats for executables and load libraries. As the code runs in the execution space, we monitor behavior in order to provide feedback on how well particular transformations are working. If a particular key results in an uncommon number of addressing exceptions or segment violations we can modify our key generation to avoid such transformations.

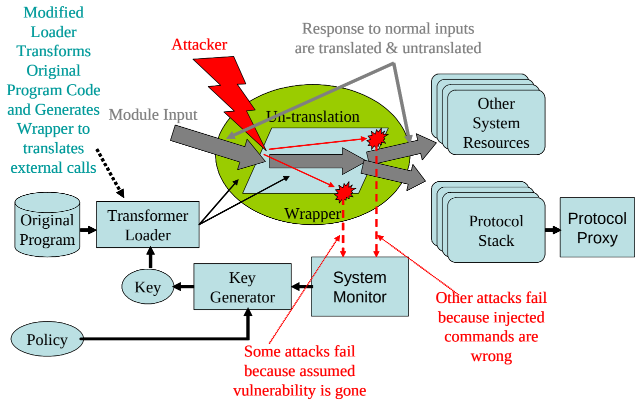
Modified Loader Transforms Original Program Code and Generates Wrapper to translates external calls
Attacker
Response to normal inputs are translated & untranslated
Module Input
Un-translation
Other System Resources
Wrapper
Original Program
Transformer Loader
Protocol Stack
Protocol Proxy
Key
Key Generator
System Monitor
Other attacks fail because injected commands are wrong
Policy
Some attacks fail because assumed vulnerability is gone

Figure 4. Diversity Architecture in Use

The diagram in figure 4 illustrates how the functional modules described above would interact to allow normal execution and to thwart injected attacks. For simplicity, the annotations, transformer, and loader are shown in this diagram as one module, the Transformer-Loader. That module transforms the original stored program and generates an associated wrapper, if needed, to retranslate external calls. The transformed program is effectively in another language that executes on the same hardware but whose assumptions about the resources and how to interact with their environment are distinctly changed. Not only are they changed, but they are changed unpredictably based upon a random key specific to this instantiation of the program. Users' inputs arrive via the normal channels. Accesses to other system resources from the transformed code are untranslated appropriately so they still work. Injected code (e.g. crafted binary data which the attacker has managed to execute) will encounter a very unfamiliar environment. Some attacks will fail because the assumed vulnerability is gone. Others will fail because the injected commands do not find the system resource names they need. With high probability, attacking code will simply fail and crash the process or endless loop which increases the detection likelihood.

One difficulty with binary level analysis is that some higher level abstractions, which are readily apparent in source code, can be difficult to reconstruct reliably at the binary level. This is essentially what obfuscation is all about. To mitigate such difficulties, some human analysis and annotation can be used to supplement automated analysis. The resulting annotation files enable the transformer to apply more sophisticated transformations than would otherwise be possible.

We are also contemplating integrating other approaches that focus on source code transformation and simple load-time transformations (not involving binary level analysis). These two approaches are quite complementary. For example, protecting data from memory error attacks is very difficult and usually involves randomizing the order and relative distance between code or data objects. It is problematic to perform these kinds of relocations at runtime on many binaries because of the difficulty of distinguishing pointers from non-pointers and code from data or object sizes. It is very easy for compilers to generate annotations to binaries so that the relocations can be done safely and easily at runtime. This is a very useful and powerful combination of approaches.

6. EXAMPLE ATTACK AND POTENTIAL DEFENSES

Most of the recent large-scale cyber-attacks have been worms that relied on buffer overflow vulnerabilities in widely deployed Windows applications. The current Windows monoculture gives even casual attackers, armed with buffer overflow attack code, the potential to break into essentially every vulnerable computer connected to the network. This enormous reach provided to the attacker can adversely affect the network itself, overloading links and cause routing table instabilities.

The Code Red I worm has been studied extensively. It is a single http GET transaction that exploits a buffer overflow vulnerability in the Index Server resource and then executes its specific payload (it could, in fact, be any arbitrary code). The following selection of disassembled code was taken from an eEye Digital Security report on the Code Red Worm [34]. It illustrates some of the common techniques and assumptions used by many worm writers. The disassembled code is on the left of the listing below and the eEye Digital Security annotations to each line are on the right. The technique illustrated below is called RVA (Relative Virtual

Address) lookup[2]. Note that Lines 09 through 13 below have no net effect and are only used for padding.

```
01  loc_4B4:         ; CODE XREF: DO_RVA+26D↑j
02  mov  esi, esp
03  mov  ecx, [ebp-198h]; set ecx with the data segment
     pointer
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
08  cmp  esi, esp         ; Compare Two Operands
09  nop         ; No Operation
10  inc  ebx         ; Increment by 1
11  dec  ebx         ; Decrement by 1
12  inc  ebx         ; Increment by 1
13  dec  ebx         ; Decrement by 1
14  mov  ecx, [ebp-1B4h]      ; load ecx with ebp-1b4
15  mov  [ebp+ecx*4-174h], eax ; load the address into
     the ebp stack where needed
```

Once RVA techniques are used to get the address of *GetProcAddress*, *GetProcAddress* is used to get the address of *LoadLibraryA* (not illustrated above). Between these two functions, all other functions that the worm may need can be easily found. The worm uses these two functions to load *kernel32.DLL*, *infocomm.DLL* and *WS2_32.DLL* enabling it to access the file system, open network sockets and send and receive network packets.

The worm is making many assumptions about how to interact with the Windows operating system and these assumptions are sensitive to particular types of code transformations. Consider the following mechanical transformations:

- 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

Any one of these transformations could break Code Red. For example, if we changed the order of parameters to GetProcAddress, the order of the push operations at lines 4 and 6 would be invalid. That would have prevented Code Red from resolving the addresses it needs to call socket, connect, send, recv, TcpSockSend, etc. When the call at line 7 is performed, the call would fail with high probability because of an invalid opcode or addressing exception as it attempts to execute the relative virtual address table.

7. CONCLUSIONS AND NEXT STEPS

Program diversity through program transformation harnesses powerful theoretical techniques to introduce much needed diversity into modern networks. We believe the potential for practical application of these techniques is high. We have described computer vulnerabilities and how attackers exploit them successfully. Our concept for a diversity system focuses on breaking the vulnerability specifications that successful attacks depend on. We extensively reviewed the types of transformations that are available for source and executable code and analyzed what we know about their effectiveness and impacts. We concluded with a functional architectural design of a diversity system based on these concepts and have anticipated some of the implementation difficulties. We will report on the success of our approach in the future.

8. REFERENCES

[1] Aleph One, “Smashing The Stack For Fun And Profit”, Phrack 49, Volume Seven, Issue Forty-Nine, File 14 of 16, 11/8/1995

[2] 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

[3] A.Avizienis, “N-Version Approach to fault tolerant Software”, IEEE-Software eg., vol- SE11, No12, Dec 1985, pp.1491 -1501

[4] 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.

[5] R. Balzer, N. Goldman. Mediating Connectors. Proceedings of the 19th IEEE International Conference on Distributed Computing Systems, Austin, Texas, May 31-June 4, 1999, IEEE Computer Society Press 73-77

[6] 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.

[7] 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.

[8] V. Bharathi, “N-Version programming method of Software Fault Tolerance: A Critical Review”, Indian Institute of Technology, Kharagpur 721302, December 28-30, 2003

[9] Sandeep Bhatkar, Daniel C. DuVarney, and R. Sekar, “Address Obfuscation: An Efficient Approach to Combat a

[2] Basically this means that all functions, and specifically GetProcAddress, are found within IIS itself. For more details on RVA, check documentation on Portable Executable (PE), the executable file format for Microsoft platforms, or read through the assembly code of this worm in the reference.

Broad Range of Memory Error Exploits," 12th USENIX Security Symposium, August 2003.

[10] L. Chen and A. Avizienis, "N-version programming: A fault-tolerance approach to reliability of software operation," IEEE 8th FTCS, pp. 3-9, 1978

[11] M. Chew, D. Song. "Mitigating Buffer Overflows by Operating System Randomization," Technical Report CMU-CS-02-197.

[12] 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

[13] 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)

[14] C. Collberg, C. Thomborson, and D. Low. "A Taxonomy of Obfuscating Transformations". Technical Report 148, Department of Computer Science, University of Auckland, July 1997.

[15] 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

[16] 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.

[17] 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

[18] 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.

[19] 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.

[20] 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

[21] 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

[22] James E. Just, et al., "Learning Unknown Attacks A Start," Recent Advances in Intrusion Detection, 5th International Symposium, Zurich, Switzerland, October 16-18, 2002, Proceedings, A. Wespi, G. Vigner, and L. Deri, (Eds.), Springer, Lecture Notes in Computer Science.

[23] 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.

[24] 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.

[25] 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.

[26] 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

[27] 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.

[28] 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

[29] D. L. Lough. A Taxonomy of Computer Attacks with Applications to Wireless Networks. PhD Thesis, Virginia Polytechnic and State University, Blackburg, VA.

[30] Douglas Low, Java Control Flow Obfuscation, MS Thesis, Univ. Auckland, 3 June 1998

[31] 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.

[32] Mudge, "How To write buffer overflows", http://www.insecure.org/stf/mudge_buffer_overflow_tutorial.html, 10/20/1995

[33] Pax. Published on World-Wide Web at URL http://pageexec.virtualave.net, 2001.

[34] Ryan Permeh, Marc Maiffret, Code Red Disassembly Analysis, eEye Digital Security, http://www.eeye.com/html/advisories/codered.zip.

[35] B.Randell, "System structure for Software Fault Tolerance," IEEE- Software Eng.,vol. SE-1,pp.220-232, June 1975.

[36] Jeff Rowe, "Diversity Draft", private communication, UC Davis, 25 Nov. 2003

[37] 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

[38] Stuart Staniford, Nicholas Weaver, Vern Paxson. “Flash Worms: Is there any Hope?” Silicon Defense, Retrieved 27 March 2003 <http://silicondefense

[39] 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>.

[40] Chenxi Wang, “A Security Architecture for Survivability Mechanisms.” PhD thesis, University of Virginia, October 2000.

[41] Chenxi Wang, "Protection of software-based survivability schemes", in the proceedings of 2001 Dependable Systems and Networks. Gutenburg, Sweden. July 2001.

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

[43] Gregory Wroblewski, “General Method of Program Code Obfuscation,” PhD Dissertation, Wroclaw University of Technology, Institute of Engineering Cybernetics, 2002.

[44] 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

[45] 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