MODELING SECRETLESS SECURITY IN N-VARIANT SYSTEMS

A Thesis
Presented to
the faculty of the school of Engineering and Applied Science
University of Virginia

In Partial Fulfillment

of the requirements for the Degree

Master of Science in Computer Science

by

Eric Weatherwax

May 2009

APPROVAL SHEET

The thesis is submitted in partial fulfillment of the

requirements for the degree of

Master of Science in Computer Science

____
Eric Weatherwax

This thesis has been read and approved by the examining Committee:

____
Thesis advisor

____

____

Accepted for the School of Engineering and Applied Science:

____
Dean, School of Engineering and
Applied Science

May 2009

Abstract

The N-variant architecture combines redundancy and tailored diversity to provide secretless security for a computer system. This thesis presents a new model of N-variant systems. This model generalizes previous work and establishes constructive guidelines for future N-variant systems. The model characterizes the components of N-variant systems and identifies the properties they have. In particular, four vulnerability types are revealed by the model that all N-variant systems should address explicitly. The model also serves as a guiding framework for building new N-variant systems. An N-variant construction process is outlined as a sequence of logical steps guided by the structure and characteristics given by the model.

The new N-variant model enables a detailed analysis of N-variant systems. The model is evaluated by analyzing five existing N-variant systems with respect to the structure given by the model and the four vulnerability types. An instantiation of the model is shown for each system as it was derived from the general model. These instantiations along with N-variant component mappings show how each system fits the general model. Additionally, each system is analyzed in terms of the four vulnerability types, using the model to show the flaws in each system.

The N-variant construction process is evaluated by presenting an experiment in designing and implementing a new N-variant system. This experiment follows the construction steps to build a 2-variant system that protects information stored in a

database. The new system is evaluated according to its effectiveness in achieving a desired security property, its consideration of the four vulnerability types, and a comparison to similar techniques. The results of the experiment produce a system that provides the desired security property without relying on secrets.

List of Figures

Figure 1. Address Space Partitioning 5
Figure 2. Traditional System....................15
Figure 3. High-Level 2-Variant System 17
Figure 4. Data Source Structure....................20
Figure 5. Data Sink Structure....................24
Figure 6. General N-Variant Model 31
Figure 7. Address Space Partitioning Source and Sink 42
Figure 8. Address Space Partitioning Model 43
Figure 9. Address Shifting Example 48
Figure 10. Instruction Set Tagging Source and Sink....................51
Figure 11. Instruction Set Tagging Model....................52
Figure 12. UID Variations Source and Sink 56
Figure 13. UID Variations Model 57
Figure 14. Dynamic Reexpression in UID System Data Sinks....................59
Figure 15. Buffer Overflow Attack 63
Figure 16. Stack Reversal Source and Sink 64
Figure 17. Stack Reversal Model 64
Figure 18. SQL System Original Architecture....................72
Figure 19. Data Sink Software Layer 77
Figure 20. SQL N-Variant Model 82

Table of Contents

Chapter 1: Introduction 1

Chapter 2: N-Variant System Overview 4

Chapter 3: Related Work 7

3.1 Diversity Techniques 7

3.2 Design Diversity 8

3.3 Redundant Execution 10

3.4 Existing N-Variant Systems 11

3.5 N-Variant System Analysis 12

Chapter 4: N-Variant Model 15

4.1 Traditional System Structure 15

4.2 N-Variant System Structure 17

4.3 Input Splitter 19

4.4 Data Sources 20

4.5 Data Sinks 23

4.6 Monitor 26

4.7 Output Merger 27

4.8 Effects and Treatment of Vulnerabilities 28

4.9 Model Summary 30

4.10 Model Contributions 31

Chapter 5: N-Variant Construction 33
5.1 N-Variant Synthesis Steps..33
Chapter 6: Analysis of N-Variant Systems..41
6.1 Analysis of Address Space Partitioning..41
6.2 Analysis of Enhanced Address Space Partitioning..47
6.3 Analysis of Instruction Set Tagging 51
6.4 Analysis of UID Variations..56
6.5 Analysis of Stack Reversal..63
6.6 Summary of Vulnerabilities 68
Chapter 7: SQL N-Variant System..71
7.1 System Model..71
7.2 Design Process..72
7.3 SQL N-Variant Model 82
7.4 Assessment 83
7.5 Vulnerability Analysis 86
7.6 Comparison to Other Techniques 87
Chapter 8: Conclusion 91
Bibliography..93

Chapter 1: Introduction

This thesis presents a new model of N-variant systems. N-variant systems previously have been modeled by considering the sequences of program states among variant processes [16] and by considering a series of hierarchical interpreters [29]. These models provided a basic framework for reasoning about N-variant systems and offered some insights into their effective use, but they did not provide detailed examinations of some aspects of N-variant systems. The model presented in this thesis provides a general view of the architecture and functional components shared by N-variant systems. This model reveals new properties and vulnerabilities not previously considered. This model also provides insights for establishing the process of designing and implementing new N-variant systems.

An N-variant system can guarantee certain security properties without requiring a secret such as a password, an encryption key, details of the software in use, or the system architecture [9, 16, 29, 33]. This characteristic is referred to as *secretless security*. The value of secretless security is considerable, because it eliminates a serious practical difficulty, keeping a secret. The ability to design systems with security properties that are secretless is attractive, and that was the primary motivation for development of a general N-variant systems model. The model allows developers to design N-variant systems starting with desired security properties and the security goal of enforcing those properties using secretless security.

The security properties of a specific N-variant system derive from the components and architecture of that system. The N-variant systems model shows how security properties derive from the system architecture. The model generalizes previous work and establishes constructive guidelines for building an N-variant system based upon the security properties desired for that system. In addition, the model identifies four types of potential vulnerabilities to which all N-variant systems are subject and which all N-variant systems must address.

Previous work introduced N-variant system concepts and components. That work helped to establish N-variant system properties, but the results did not facilitate the systematic design of an N-variant system determined by a set of desired security properties. The model provided here establishes properties for N-variant systems and permits their analysis. The model also provides a link between desired properties and the N-variant architecture needed to achieve them, thereby serving as a guide from which to create new N-variant systems.

The main contributions of this thesis are:

- A general model of N-variant systems, detailing the components, data flows, properties, and vulnerabilities.
- A process for constructing new N-variant systems.
- An analysis of existing N-variant systems with respect to the model. The model enables and guides the analysis of these systems to determine the vulnerabilities to which they are susceptible.
- An experiment in creating an SQL N-variant system, using the construction process and the model as guides.

The next chapter reviews the N-variant system concept and recalls an earlier system based on partitioning address spaces as an example. Chapter 3 discusses related work. Chapter 4 presents the general N-variant systems model, describing the system structure, the set of components, and potential vulnerabilities. Chapter 5 evaluates the model's ability to guide the construction of N-variant systems by outlining an N-variant construction process based on the model. Chapter 6 evaluates the validity of the model by examining existing N-variant systems, using the model to identify components and analyze properties and vulnerabilities. Chapter 7 presents an experiment that evaluates the N-variant construction process by building a system designed to protect an SQL database. Chapter 8 concludes the thesis.

Chapter 2: N-Variant System Overview

The N-variant architecture combines redundancy and diversity to secure a computer system. Artificial diversity is applied to two or more (i.e., N) instances of the same system to create system variants that are then executed with identical input data and under the supervision of a monitor. Execution is usually synchronized and in parallel, although it does not have to be. The diversity is tailored so as to make it impossible to compromise all variants simultaneously with the same input.

Tailored diversity allows N-variant systems to provide guarantees against specific attack classes. Strong arguments can be made with respect to the protected attack class, regardless of the vulnerability being exploited. As a result, known vulnerabilities do not need to be eliminated in order to maintain protection. The disjointedness of the diversity applied to the variants means that an attack (from the class for which protection is offered) which exploits one of the variants will not have the same effect on any other variant. With an appropriate monitor in place, any of these attacks will be detected.

The artificial diversity applied to system variants is *data diversity* [1]. Data diversity changes the representation of information from its original form to a new form using a process called *data reexpression*. Provision must be made in any component that processes the data to operate correctly with the reexpressed data. One approach is to reverse the reexpression before the data is used, and a second approach is to modify the component to work with the reexpressed data. Data diversity can be applied to any data

stream within a system. In an N-variant system, data diversity is applied specifically to the data streams that are vulnerable to manipulation by attacks.

In a conventional system, an attack exploits a vulnerability in one part of the system to pass malicious data to another part of the system. An N-variant system provides protection against a given attack because there are always at least two variants that must behave identically but use diverse internal data streams. The internal data streams have been subject to different data reexpressions, and so a successful attack requires that *multiple* diverse data streams be modified by *one* malicious payload.

The above requirement gives N-variant systems a significant advantage over traditional diversity. Instead of relying on a single form of unknown diversity, N-variant systems rely on the difficulty of manipulating differently diverse data streams with a single payload. This allows an N-variant system to provide protection without relying on assumptions about keeping secrets. There is no secret key or algorithm that, if revealed, will compromise the security of the system. In fact, the reexpression functions and the variants themselves can be fully published without any loss of security.

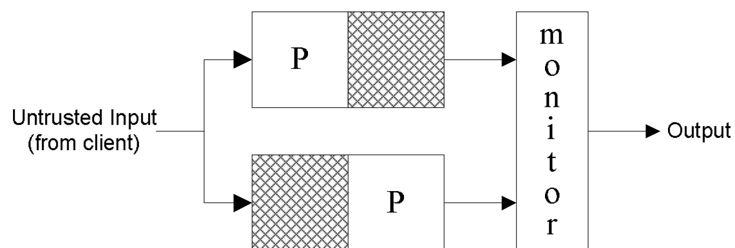

Figure 1. Address Space Partitioning

Figure 1 illustrates an example N-variant system using address space partitioning. In this system, the two variants are derived from a program P with addresses relocated such that a valid absolute memory reference in one variant is guaranteed to be invalid in

the other. For this system, the data that was reexpressed was the program binary, and the applied reexpression changed the representation of valid absolute addresses.

Consider a memory overwrite attack that exploits a vulnerability in a program in order to pass malicious data to the hardware that is executing the program. A 2-variant address space partitioning system was constructed to thwart memory overwrite attacks [16]. In such a system, two variants execute in disjoint memory regions. On normal (non-malicious) input, the system behaves as it did originally. However, consider malicious input that gains access to memory and overwrites a memory location using an absolute address. Such an attack might operate as the attacker desired in one variant, but will not do so in the other variant because the memory regions are disjoint and the address will be out of bounds.

Figure 1 is useful for illustrating the basic concept of N-variant systems, but the figure displays only a summary of the N-variant system architecture. The figure fails to show many of the integral N-variant components. The model presented in Chapter 4 incorporates all of the necessary components of N-variant systems and explores the general architecture.

Chapter 3: Related Work

This chapter describes work related to N-variant systems. These works are organized into five categories: (1) Diversity Techniques; (2) Design Diversity; (3) Redundant Execution; (4) N-Variant System Architectures; and (5) N-Variant System Analysis.

3.1 Diversity Techniques

Diversity techniques apply transformations to software to increase the difficulty of exploiting vulnerabilities. N-variant systems use multiple forms of diversity that have been carefully constructed to produce disjoint data.

Diversity has been applied to the layout of memory to protect against memory errors [7, 18, 19, 46]. Xu et. al proposed Transparent Runtime Randomization (TRR) to relocate the stack, heap, shared libraries, and runtime control data structures in an application's memory space [46]. Their implementation modified the Linux dynamic program loader to effect the randomized relocations. Bhatkar et. al used a source code transformer to randomize absolute locations of code and data objects as well as their relative distances [7].

Diverse instruction sets have been proposed to protect against code injection attacks. Kc et. al used a modified Linux kernel and an x86 emulator to create process-

specific randomizations of the x86 instruction set [25]. Barrantes et. al used a binary translator to randomize instructions for programs executing on a virtual machine [4].

Chew and Song proposed three diversity techniques for mitigating buffer overflow attacks [14]. They randomized system call mappings, library entry points, and the stack starting point, noting that each of these must necessarily be known by an attacker in order to execute a buffer overflow attack and successfully interact with the operating system.

Other diversity techniques have been applied to randomize compiler layouts [1, 19], encrypt pointers [15, 42], and randomize SQL [8]. All of the discussed approaches rely on the secrecy of keys used to apply the diversity. The afforded protection depends on high entropy of the key space, but this assumption might not be valid for certain diversity techniques. Shacham et. al used a derandomization attack to exploit systems protected by address-space randomization (ASR) [36]. Sovarel et. al used an analogous approach to exploit systems protected by instruction set randomization (ISR) [38]. Unlike these traditional diversity techniques, N-variant systems do not rely on any such secrets keys and can potentially employ any diversity technique, including those with low entropy.

3.2 Design Diversity

The N-variant systems concept was motivated in part by the ideas of design diversity. Design diversity involves the independent development of multiple versions of the same program. It has been used previously as a means for achieving fault tolerance in software systems [28].

N-version programming is a technique based on design diversity that uses multiple independent teams developing software which implements the same set of requirements [3, 12]. The N-version programming approach is to execute the independent software versions in a framework which allows their outputs to be examined by a decision function. The conjecture is that independent teams will introduce independent faults in their software, thereby avoiding common faults among the versions. However, Knight and Leveson conducted experiments showing that this assumption of independence might not be valid [26].

Joseph proposed using N-version programming as a way to secure a computer system [23]. He analyzed N-version programming to see if it could be used to defeat security attacks. He developed an analogy between faults that affect reliability and vulnerabilities that affect security. Using this analogy, he concluded that N-version programming might allow vulnerabilities to be masked in the same way traditional N-version programming masks faults.

Littlewood and Strigini discuss the roles and limits of design and data diversity in security [27]. They argue the need for probabilistic reasoning about security and show that insights from probabilistic modeling in reliability and safety apply to the use of design diversity for security. They present an analysis of the security properties for systems using traditional design diversity.

N-variant systems do not rely on the statistical probabilities relied on by design diversity frameworks. Instead, variants are engineered to differ in specific ways that facilitate attack detection. Additionally, the variants are constructed mechanically from

the same program versions, and so multiple development teams and their associated costs are not required.

The HACQIT project [24, 32] introduced an intrusion tolerant system architecture for providing Internet services to known users. HACQIT used multiple implementations of the same protocol to gain the benefits of N-version programming without the disadvantages associated with the costs of independent development teams. This was termed COTS (Commercial Off-The-Shelf) diversity. HACQIT deployed two existing web servers (IIS on Windows and Apache on Linux) and compared their HTTP status codes to detect an intrusion. The challenge of COTS diversity is in handling different outputs that might represent design differences in the products but are otherwise benign.

Totel et. al [41] and Gao et. al [21] present similar COTS diversity architectures. Totel et. al compare actual web page responses from web servers to determine divergent behavior. Gao et. al measure behavioral distance using a metric based on the sequence of system calls each server made.

3.3 Redundant Execution

Architectures have been proposed that execute redundant versions of a program for the purposes of dependability and fault tolerance. This work has motivated in part the redundant diversity framework of N-variant systems.

Berger and Zorn proposed DieHard, a runtime system that provides probabilistic memory safety. DieHard executes redundant versions of multiple variants, with each variant diversified with a randomized heap layout. The DieHard framework was limited

to monitoring variants only through standard I/O. The primary goal of DieHard was not to prevent attacks and improve security, but rather to enhance system reliability.

Schneider and Zhou suggested proactively diversifying program replicas as a means of hiding design flaws in replicated software [34]. Their approach was to periodically randomize replicas to limit the window of vulnerability when the replicas are susceptible to the same attacks. This approach is complementary to the N-variant systems approach in that diversity is applied to redundant variants.

3.4 Existing N-Variant Systems

Cox et. al introduced N-variant systems as a framework for using automated diversity to thwart entire classes of attacks [16]. Their work presented a model of execution for an N-variant system by considering the sequences of program states among variant processes. They demonstrated two N-variant systems: one which diversified memory addresses (address space partitioning) and one which diversified program code (instruction set tagging).

Bruschi et.al extended the address space partitioning system to thwart partial memory overwrites, which were a vulnerability in the original address space partitioning system [9]. They also addressed some of the practical issues of implementing N-variant systems. They explored methods of handling shared memory in the variants and handling non-determinism that can affect the synchronization of variants.

Nguyen-Tuong et. al employed data diversity in an N-variant system to protect against the corruption of important user identification data [29]. The paper presented an interpreters model for reasoning about how data transformations preserve the necessary

N-variant properties. The implemented system required a modification of program code rather than the transformation of program binary seen in earlier examples.

Franz argued for the use of N-variant systems to counter the malicious insider threat in software development [20]. Franz proposed running two variants with reversed stack orderings in order to prevent stack smashing attacks. He noted that this framework would even defeat attackers who had purposefully inserted vulnerabilities into the program (i.e. insiders). Franz also suggested that multi-core processors can reduce the performance impact of running redundant versions a program.

Salamat et. al described a multi-variant execution environment (MVEE) in which several versions of the same program are executed in lockstep [33]. The MVEE framework had similar characteristics to other N-variant systems, except that it did not rely on operating system modifications, similarly to the enhanced address space partitioning system [9]. An MVEE named Orchestra was implemented using reverse stack growth directions.

3.5 N-Variant System Analysis

Cox et. al modeled N-variant systems as a framework composed of N variants, a monitor, and a polygrapher [16]. The polygrapher copied input to all N variants and the monitor observed the behaviors of the variants to detect a divergence. This work modeled the execution for an N-variant system as a sequence of the variant states. Each variant was considered to have three possible states: normal, compromised, and alarm. This work also described two system properties necessary for obtaining system correctness and the desired attack detection. A normal equivalence property stated that

whenever all variants are in normal states, they must be in states that correspond to the same canonical state. A detection property stated that whenever one variant enters a compromised state, at least one other variant must enter an alarm state.

Nguyen-Tuong et. al used an interpreters model to reason about data diversity in N-variant systems [29]. They considered applications to be composed of a series of hierarchichal interpreters, each processing a particular type of data. Attacks were modeled as exploiting vulnerabilities in a high-level interpreter in order to control inputs to a target interpreter. Data diversity changed the interfaces between interpreters. Using this model, the normal equivalence property is established by ensuring that all trusted data is transformed and operations on that data are transformed to preserve the original semantics. The detection property is established by showing that the reexpression functions used to apply the data diversity are disjoint.

Salamat et. al modeled multi-variant execution environments (MVEE) as a Monitor process being the main component that executes variants as child processes [33]. The Monitor ensures functional equivalence with the original system by performing all I/O operations itself and sending identical results to the variants. The monitor uses the invocation of system calls as variant synchronization points. The variants are considered to be in conforming states if all variants invoke the same system call with identical sets of arguments within a pre-defined window of time.

The N-variant model presented in this thesis differs from these models in the way the variants are modeled and how the model can be used. Each variant is modeled as a data source that passes reexpressed data to a corresponding data sink. Static and dynamic reexpression mechanisms are explicitly distinguished in the data sources and data sinks.

This structure facilitates the identification of component characteristics and the identification of four vulnerability types. Finally, the model has facilitated the derivation of a synthesis process from which additional N-variant systems can be constructed to provide prescribed security properties.

Chapter 4: N-Variant Model

This chapter presents the general N-variant system model. This model characterizes the components of an N-variant system, the properties of these components, and the data that flows between them. The model also reveals vulnerabilities that might exist within N-variant systems.

4.1 Traditional System Structure

Before discussing the general N-variant model, it is important to understand the model of a traditional system before N-variant security is applied. This model illustrates how components and data are mapped from a traditional system to its N-variant form. Figure 2 shows the structure of a traditional system in which the vulnerable component, referred to as the *data source*, is shown connected to a single separate component, referred to as the *data sink*.

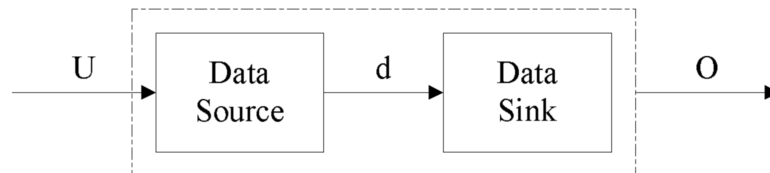

Figure 2. Traditional System

This figure is not intended to model the design or functional units of a system, but rather it models the way in which data flows from one part of the system to another. The

data source and data sink are not necessarily distinct processes or program modules, but they are data processing pieces of the system. Any system can model its data streams with a piece that generates the data and a piece that processes the data. This producer and consumer data relationship is what Figure 2 models.

The data source receives external input for the system, labeled ‘U’ in the figure. After processing the input, the data source produces data that is then consumed and processed by the data sink. This data is labeled ‘d’ in the figure. Both the data source and data sink may produce output that is consumed by components external to the source/sink pair. This system output is labeled ‘O’ in the figure and is shown leaving a dashed line box, which represents the combined source/sink system.

As examples of how this structure models real systems, consider the following: A program executing on a machine produces addresses that are used to access the contents of memory. The data are the addresses, the data source is the program, and the data sink is the machine’s physical memory. As another example, a web application uses SQL queries to retrieve data from a database. The data are the SQL queries, the data source is the web application, and the data sink is the database.

The data sink is the target of attacks of interest. The mechanism of an attack of interest is to exploit a vulnerability in the data source in order to manipulate the data being passed from the data source to the data sink. Because the data sink expects data of a particular form, an attacker can manipulate the data such that the data sink is compromised in a specific malicious way.

As an example, consider a buffer overflow code injection attack. The data source is the program that created the buffer and the data sink is the machine hardware. An

attacker exploits the buffer overflow vulnerability in the program in order to manipulate the instructions that will be executed on the machine.

4.2 N-Variant System Structure

The N-variant architecture introduces N source/sink pairs that each operates on data of a different reexpressed form. This architecture combats attacks of interest, provided that the attack mechanism is to manipulate the data being passed from the data source to the data sink ('d' from Figure 2). The term N-variant refers to the N source/sink pairs that replace the original source/sink pair. Each pair is referred to as a variant. In general there can be N different variants in an N-variant system. At least two variants are required to achieve the diversity that makes N-variant systems effective. There is no maximum number of allowable variants, but the more variants there are, the more resources are required to execute them. Figure 3 shows an N-variant system consisting of 2 variants derived from the traditional system of Figure 2.

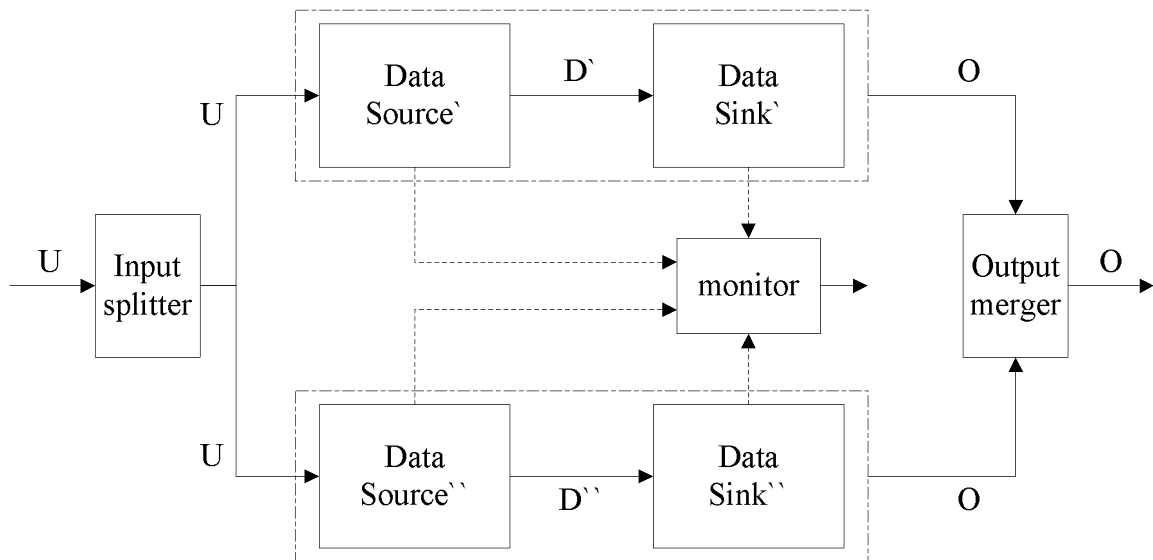

Figure 3. High-Level 2-Variant System

As Figure 3 illustrates, an N-variant system is composed of five components: (1) an input splitter; (2) the data sources; (3) the data sinks; (4) a monitor; and (5) an output merger. The functionalities of the components are:

Input Splitter: The input splitter replicates and routes inputs, labeled U in Figure 3, to the data source for each variant.

Data Sources: Each data source implements part of the system functionality. The data source is supplemented in various ways to support the necessary data reexpression.

Data Sinks: Each data sink also implements part of the system functionality. The reexpressed output data produced by each data source, labeled D` and D`` in Figure 3 is passed to the corresponding variant's data sink.

Monitor: In the N-variant framework, processing of the reexpressed data by the data sinks will result in divergent behavior on abnormal (attack) inputs. The monitor observes the behavior of both data sinks and raises an alarm when their behaviors diverge. A divergence might be manifested as information from the data sinks that differs or execution states of the data sinks or data sources that differ.

Output Merger: The output merger combines redundant outputs from the N variants into single system outputs. These outputs may come from any part of the variants, including the data sources and data sinks. Output merging is necessary so that the entire N-variant system behaves as a single logical entity from the point of view of any clients of the system.

The next five sections discuss each of the N-variant components in further detail. The five components have security characteristics that apply to all instances of N-variant systems. In addition to a more detailed description of each N-variant component, each

section identifies these necessary security characteristics and explains how the component can differ across N-variant implementations.

4.3 Input Splitter

Input in an N-variant system is defined as any data that originates from outside the system. All inputs are considered untrusted and might contain malicious payloads. By these definitions, input is usually referring to user supplied data. An N-variant system may have multiple inputs coming from multiple sources, but only one input stream is shown in Figure 3 to simplify the model. No generality is lost by doing this because the model does not constrain the number or types of input to the system. Additionally, all security properties pertaining to input or the input splitter will apply to every source of input to the N-variant system. Note that while number and types of input to the N-variant *system* are not constrained, the input data supplied to the *data sources* from the input splitter must be identical.

The input splitter is the first component of an N-variant system. The input splitter's function is to accept input for the system and to replicate and route that input to the data source for each variant. The necessary characteristics of the input splitter are that it does not alter the input data, the order of the data it sends to the variants is the same as the order of the inputs it received, the data it sends to all variants is identical, and it sends data to all data sources such that their execution states do not diverge.

4.4 Data Sources

The second component of an N-variant system consists of the data sources. Each data source in an N-variant system is created by modifying the original data source so that it applies diversity to its output data stream. The form of data diversity used depends on the security requirements. For example, if the system is meant to preserve the integrity of user identification data, then the data sources can be constructed to apply diversity to all data representing user identification values. The same reasoning can be applied to N-variant systems with other security requirements.

Figure 4 shows the structure of a data source. Each data source is modeled as five segments: (1) input data, labeled U; (2) output data, labeled D; (3) software vulnerabilities, labeled I, II, and III; (4) the output location, labeled X; and (5) reexpression mechanisms, labeled Rex_D and Rex_S. The next four subsections provide further detail on these data source segments.

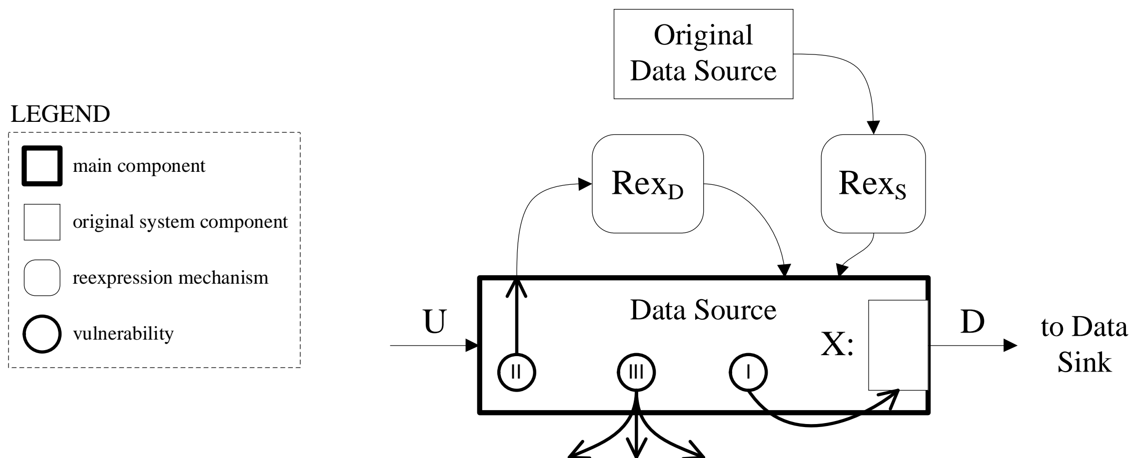

Figure 4. Data Source Structure

4.4.1 Input and Output Data

The input data U is provided by the input splitter and is derived from external sources. The input data is inherently untrusted and might contain malicious payloads. The output data D is sent to the target data sink. The output data is reexpressed and must have a different form for each data source.

4.4.2 Output Location

The output location is the location where output is last held by the data source before being passed to the data sink. The output location helps model the way in which data is written to the data channel between source and sink. An attack either directly or indirectly overwrites the output location in order to manipulate the data being passed from data source to data sink.

4.4.3 Vulnerabilities

Any malicious payload will target software vulnerabilities in the data sources in an effort to control the input to the target data sinks. Vulnerabilities are illustrated as circles in Figure 4 that contain roman numerals indicating the vulnerability type.

- Type I vulnerabilities allow an adversary to overwrite the output data of the data source. An attacker exploiting a Type I vulnerability can control the data source's output to a data sink.
- Type II vulnerabilities allow an adversary to use the data source's reexpression mechanism. An attacker exploiting a Type II vulnerability can correctly reexpress malicious data in all data sources.

- Type III vulnerabilities allow an adversary to use the data source in order to interfere with other components of the system.

These three vulnerability types are discussed in greater detail in Section 4.8.

4.4.4 Reexpression Mechanisms

Each data source uses a reexpression function to apply diversity to the relevant information being protected. These reexpression functions are referred to as R` for the first variant, R`` for the second variant, and so on. The reexpression functions are not explicitly shown in the data source diagram, but instead the reexpression mechanisms that apply these functions are shown. The reexpression mechanisms are labeled as Rex_D and Rex_S in Figure 4.

Reexpression can be applied dynamically at runtime, statically by encoding the reexpressed information into the data source program, or by a combination of the two. Rex_D and Rex_S represent the dynamic and static reexpression mechanisms, respectively.

Rex_D is a runtime function that applies the appropriate reexpression function in order to convert relevant data to a reexpressed form. Rex_S represents a transformation of the original data source program. Rex_S is a static transformation that modifies the data source to operate on reexpressed data. When applying the Rex_S transformation, relevant data that is hard-coded into the data source is transformed using the appropriate reexpression function. Additionally, all operations on reexpressed data are transformed to preserve the original semantics, if necessary.

As examples of static and dynamic reexpression, consider a web application source that reexpresses queries sent to a database sink by appending a '123' to SQL keywords. A static reexpression would append the '123' to hard-coded representations of

the keywords in the source code. A dynamic reexpression would append the '123' to the keywords at runtime.

4.4.5 Security Properties

The data sources must have several important properties to ensure the effectiveness of the N-variant system. They are:

- All information applicable to the desired security property must be reexpressed.
- The reexpression functions used by the data sources must be disjoint. This means that for all possible pieces of relevant information, the data sources reexpress them differently. If the reexpression functions are not disjoint, then the protection obtained is not guaranteed.
- The N data sources must be functionally equivalent and operate in semantically equivalent states on normal (non-malicious) input.

4.5 Data Sinks

The third component of an N-variant system consists of the data sinks. Each data sink in an N-variant system is created by modifying the original data sink so that it expects data of a reexpressed form from one of the data sources.

Figure 5 shows the structure of a data sink. Each data sink is modeled as three segments: (1) input, labeled D; (2) software vulnerabilities, labeled IV; and (3) inverse reexpression mechanisms, labeled Inv_D and Inv_S. The next three subsections provide detail on these data sink segments.

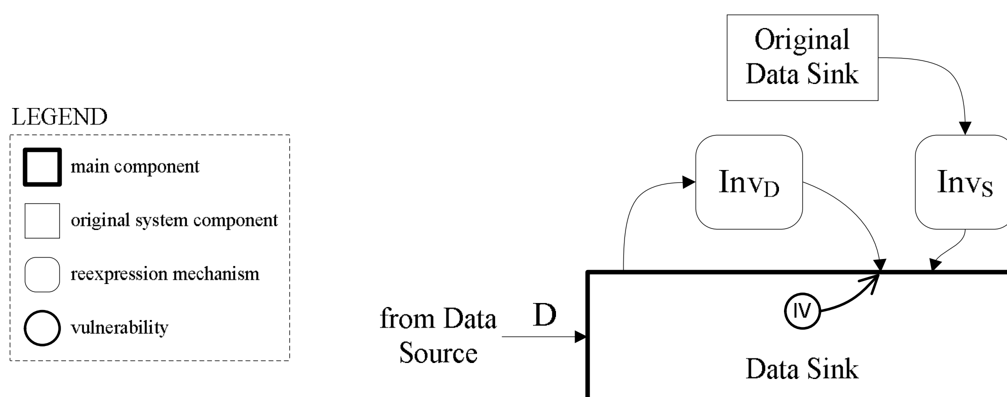

Figure 5. Data Sink Structure

4.5.1 Input

The input data D is reexpressed data from the corresponding data source. Each data sink expects reexpressed data of the form defined by its associated variant reexpression function.

4.5.2 Vulnerabilities

A fourth type of vulnerability exists in the data sink (illustrated as a circle in Figure 5 containing the roman numeral IV). Type IV vulnerabilities allow an adversary to overwrite the results of inverse reexpression. An attacker exploiting a Type IV vulnerability can circumvent the security afforded by reexpression and get malicious data into the data sink. Type IV vulnerabilities are discussed in greater detail along with the other three vulnerability types in Section 4.8.

4.5.3 Inverse Reexpression Mechanisms

Each data sink uses an inverse reexpression function to operate with the reexpressed data it receives. These inverse reexpression functions are referred to as $R^{`-1}$ for the first variant, $R^{``-1}$ for the second variant, and so on. The inverse reexpression

functions are not explicitly shown in the data sink diagram, but instead the inverse reexpression mechanisms that use these functions are shown. The inverse reexpression mechanisms are labeled as Inv_D and Inv_S in Figure 5.

A data sink can explicitly apply inverse reexpression by reversing the data reexpression of its input, or the data sink can implicitly apply inverse reexpression by constructing the data sink to operate on reexpressed data. The former is a dynamic inverse reexpression mechanism while the latter is a static mechanism for applying inverse reexpression. Inv_D represents the dynamic inverse reexpression option while Inv_S represents the static inverse reexpression option.

Inv_D is a runtime function that accepts reexpressed information as input and applies the appropriate inverse reexpression function to produce the original form as output. Inv_S represents a transformation of the original data sink program. Inv_S is a static transformation that modifies the data sink to operate on reexpressed data. Like Rex_S, any operations on reexpressed data are transformed to preserve the original semantics, if necessary.

4.5.4 Security Properties

The data sinks must have two important properties to ensure the effectiveness of the N-variant system. They are:

- The inverse reexpression functions must be inverses of their corresponding reexpression functions. This property ensures that the data sinks produce output of the same form as the output of the original data sink.

- A data sink must only accept input from its data source after reexpression has been applied. This property prevents benign non-reexpressed data from being interpreted as malicious by the data sink and causing a false alarm.

4.6 Monitor

The N-variant framework is constructed so that the variants will diverge on abnormal (attack) inputs. The fourth component of an N-variant system, the monitor, is designed to detect this divergence. The monitor observes the behavior of both variants and raises an alarm when their behaviors diverge. It can monitor for a divergence by comparing information that it receives from the data sinks, or it can monitor for a divergence by comparing execution states of the data sinks or data sources. In Figure 3, the inputs to the monitor are shown as dashed lines in order to differentiate between behavioral monitoring and the functional flow of information, illustrated as solid black arrows in other parts of the diagram. The necessary security characteristics of the monitor are that it must not inhibit the normal execution sequence of the system when no divergence is detected, and the monitor must raise an alarm when a divergence is detected.

The monitor is also responsible for initiating a recovery process upon detecting a divergence. Recovery means resetting the variants to an uncompromised state. Example recovery mechanisms can be restarting the variants or rolling the system back to an uncompromised checkpoint. Any recovery process must put the variants in uncompromised states; otherwise a future attack could succeed. In practice, this might be difficult to prove, and previous N-variant systems have implemented recovery as simply

terminating both variants. This method successfully puts the variants into an uncompromised state, but does not provide continued service. In general, there is no security requirement for the system to automatically provide continued service after a divergence, but doing so would improve the system's availability and is an area of active research.

One of the necessary characteristics of the monitor is to raise an alarm when a divergence is detected. This alarm has two parts. The first part of the alarm is initiating the recovery process outlined above. The other part of the alarm is preventing system output that derived from a compromised state. System output is any data that leaves the output merger. The divergence checking of the monitor must be fine-grained enough that it can prevent compromised output after detecting a divergence. Examples of how the monitor can prevent system output are closing communication channels, signaling the output merger, physically intercepting outgoing data, or terminating the variants.

4.7 Output Merger

The final component of an N-variant system is the output merger. The output merger is responsible for merging outputs from all variants into single system outputs. These outputs may come from any part of the variants, including the data sources and data sinks. Under normal (non-attack) conditions, the outputs from the variants will be identical, and so merging the output will be a simple case of making the redundant outputs into a single output. Under attack conditions, the monitor will signal a divergence, and the output merger should not output anything. The important characteristic of the output merger is that it produces output in such a way that the

consumer(s) of its output are not required to be modified to accommodate the N-variant version of the system. If an external entity expects only one system output, then it is the responsibility of the output merger to ensure this.

4.8 Effects and Treatment of Vulnerabilities

Recall that N-variant systems are subject to four types of vulnerabilities, identified in Section 4.4.3 and Section 4.5.2. This section provides a more detailed explanation of each vulnerability type, when each occurs, and how each affects the security afforded by N-variant systems.

4.8.1 Type I Vulnerabilities

Type I vulnerabilities allow an adversary to manipulate the output value of the data source. An attack on this vulnerability causes the data source to send unintended output to the data sink. N-variant security is designed to eliminate Type I vulnerabilities because the output location is identical in all N data sources. An attack is thus constrained to overwrite N output locations with the same malicious data. Because of disjoint data reexpression, this will always fail to be correct in at least one of the variants, and attacks on this type of vulnerability will always be detected. *N-variant systems are explicitly designed to combat Type I vulnerabilities.*

This holds true as long as the attack is corrupting whole data source outputs. If an attack can partially overwrite output data, then an N-variant system might be susceptible to partial overwrite attacks. As an example, consider the address space partitioning system in which two variants are diverse in only the highest bit of memory addresses. An attack can overwrite the other 31-bits of a memory address and it might not be detected.

Thus it is necessary for N-variant designers to make strong security claims against the effects of partial overwrites, use reexpression functions that are not vulnerable to it, or exclude partial overwrites from the attack classes covered.

4.8.2 Type II Vulnerabilities

Type II vulnerabilities allow malicious data to be reexpressed by the data source on behalf of the adversary. This can occur when the data source uses a dynamic reexpression mechanism. If an attacker can inject data into the execution sequence that uses the reexpression function, then the attacker can reexpress his malicious data correctly in all variants. Type II vulnerabilities become relevant in the decision of how the data source reexpression is done. Any dynamic reexpression is susceptible to Type II vulnerabilities, while only a completely static transformation of the data source is immune to them. The N-variant framework cannot provide any security guarantees if Type II vulnerabilities exist, and so the guaranteed effectiveness of a vulnerable N-variant system will be reduced to its security against Type II vulnerabilities.

4.8.3 Type III Vulnerabilities

Type III vulnerabilities allow malicious data to interfere with other components of the N-variant system. This vulnerability type is illustrated in Figure 4 as arrows leading out of the data source to other components of the system. Type III vulnerabilities become hazardous when components are not properly isolated. For instance, if a data source and a data sink share an address space, vulnerabilities within the data source might affect the state of the data sink. In cases like this, the designer of the N-variant system must

provide sufficient protection against Type III vulnerabilities or arguments for why they are not a security threat.

4.8.4 Type IV Vulnerabilities

Type IV vulnerabilities are present in the data sink and allow malicious data to overwrite the output of an inverse reexpression mechanism. If an attacker can overwrite this output, then the attacker can bypass the security afforded by reexpression and deliver malicious data to the data sink. Type IV vulnerabilities become relevant in the decision of how the inverse reexpression is done. Any dynamic inverse reexpression is susceptible to Type IV vulnerabilities, while only a completely static transformation of the data sink is immune to them.

4.9 Model Summary

The general N-variant system model defines a structure in which N differently diverse variants are executed in a manner that enables detection of attacks. The model contains five main components:

- Data sources are vulnerable parts of the variants that supply reexpressed data to their associated sinks.
- Data sinks are parts of the variants that process reexpressed data.
- A monitor detects divergences in the behavior of the variants that might indicate an attack.
- An input splitter replicates and routes system inputs to data sources.

- An output merger combines outputs from the variants to ensure that the system behaves as a single logical entity from the point of view of external clients.

Figure 6 shows the general N-variant model, including the main components, data flows, reexpression mechanisms, and vulnerabilities.

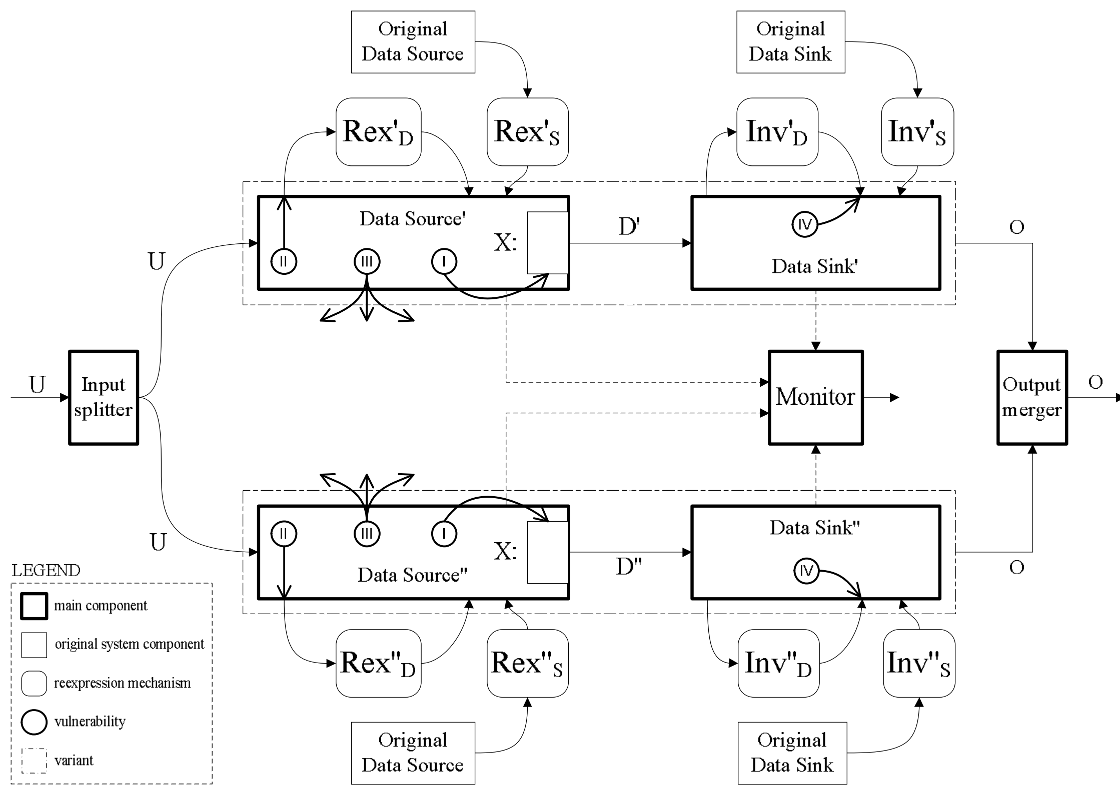

Figure 6. General N-Variant Model

4.10 Model Contributions

The model provides insights into what properties should be incorporated into the main components of an N-variant system. These insights and knowledge about the model can be used to build better systems. A process for constructing an N-variant system is presented in Chapter 5.

The components and properties elicited from the model enable a detailed analysis of N-variant systems to determine the vulnerabilities to which they are susceptible. Analyses of existing systems with respect to these vulnerability types can be used to strengthen the protection afforded by these systems. Five existing N-variant systems are analyzed in Chapter 6 in order to evaluate the model's ability to generalize N-variant systems and identify vulnerabilities.

If the model and the N-variant construction process are correct, then they can be used to implement a new N-variant system. Any system constructed using this process should provide the desired security property and address the vulnerability types identified by the model. An experiment is conducted in Chapter 7 to evaluate the effectiveness of using the N-variant construction process to implement a new system.

Chapter 5: N-Variant Construction

The N-variant model identifies characteristics and properties that are necessary for ensuring security guarantees in any N-variant system. These characteristics and properties should be built into the components from the beginning of development. This chapter outlines an N-variant construction process that uses the model to allow additional N-variant systems to be built. This process is presented as a sequence of logical steps guided by the structure and characteristics given by the model. Each step in this process must be aware of and support functional correctness, i.e., the N-variant version of the system must provide the same functionality as the original system. Section 5.1 presents the synthesis steps and guidelines for applying each step.

5.1 N-Variant Synthesis Steps

5.1.1 Step 1 – Identify desired security property

An N-variant system should be constructed by first identifying the security property desired by the system. By identifying the desired security property first, the system will be designed with a goal of ensuring that security property rather than preventing specific attacks. The designers of the system or system requirements dictate the desired security property or properties.

As an example, consider the address space partitioning system. The system designers observed that vulnerabilities allowing memory addresses to be overwritten with absolute values commonly exist within applications. The desired security property for this system was therefore ensuring that memory addresses overwritten with attacker-controlled absolute values are not accessed by the machine.

5.1.2 Step 2 – Identify information that needs to be protected

To achieve the desired security property, the system must ensure the protection of certain information. This information is identified by considering how the desired security property can be defeated. Information that is corrupted by any attack which defeats the desired security property should be protected.

Continuing with the address space partitioning example, an attack which overwrites memory addresses corrupts the contents of memory. According to the desired security property, the system should ensure that any corrupted addresses are not accessed by the machine. Thus, memory contents needed to be protected at the point in which they were interpreted by the machine.

5.1.3 Step 3 – Create the reexpression functions

Reexpression functions are created to diversify the information identified in step 2. The reexpression functions provide the tailored diversity that is central to the protection guarantees and secretless properties afforded by the N-variant framework. Creating the reexpression functions in this step will allow designers to better determine the mechanisms for applying the reexpression in later steps.

The reexpression functions take the information identified in step 2 as input and produce a reexpressed version of that information as output. The reexpression functions must be chosen so that their outputs are disjoint for all possible inputs. Additionally, the functions must be analyzed for their susceptibility to partial overwrites. If an attacker can partially overwrite reexpressed information while still allowing that information to be valid for all reexpression functions, then the attacker can effectively exploit a Type I vulnerability.

It is recommended that one of the reexpression functions be chosen as the identity function, if possible. This will greatly simplify the task of creating one of the variants as it will not need to be modified from its original implementation.

5.1.4 Step 4 – Determine inverse reexpression functions

The data sinks use inverse reexpression functions to operate with the reexpressed data they receive. Identifying the inverse reexpression functions in this step will allow designers to determine the inverse reexpression mechanisms in a later step.

The inverse reexpression functions are established by determining the inverses of the reexpression functions. After the inverse reexpression functions are determined, they must be shown to be inverses of their respective reexpression functions.

5.1.5 Step 5 – Identify data source and data sink

The data source and data sink will constitute the system variants. Identifying them now allows designers to know what pieces of the original system will need to be replicated and apply reexpression mechanisms. It is recommended to first identify the

data sink and then the data source because this allows a designer to enumerate the possible data sources with more assurance.

The data sink is identified by considering what parts of the system process the protected information. All parts of the system that process this information are included in the data sink. The data source is identified by considering which parts of the system produce the protected information. All parts of the system that produce and supply protected information to the identified data sink are included in the data source.

It is important to note that data sources and data sinks do not need to be complete programs or processes. Identifying complete program or process as a data source or data sink might make replication simpler, but it is not necessary. The stack reversal system provides an example for which the data source and data sink are both just portions of an entire program.

5.1.6 Step 6 – Determine data source reexpression mechanism and construct variant data sources

This step marks the beginning of constructing components for the N-variant system. The data sources and data sinks of the variants will be constructed first so that the other components can be created to work with implementation specific details.

The reexpression mechanism must be determined before the data sources can be created. Static transformation of the data source should always be chosen as the reexpression mechanism, if possible, because it protects the data sources from Type II vulnerabilities. However, completely static reexpression is only possible when all of the data that needs to be reexpressed is encoded in the data source. If the relevant data is obtained dynamically or from an external source, then dynamic reexpression will be

required. In these cases, a combination of dynamic and static reexpression is still preferred. Dynamic reexpression should only be used where necessary, and the data source should be statically transformed to accomplish all other reexpression.

The data sources are constructed by replicating the original data source and implementing the determined reexpression mechanism using the appropriate reexpression function for each variant. Dynamic reexpression mechanisms can be implemented as functions in the data source. Static transformation is accomplished by applying the appropriate reexpression function to all relevant data that is encoded in the data source. Additionally, the data source might need to modify operations involving the protected data to preserve the original semantics.

5.1.7 Step 7 – Determine data sink inverse reexpression mechanism and construct variant data sinks

The inverse reexpression mechanism must be determined before the data sinks can be created. The N-variant model has indicated that the data sink can be statically transformed to operate with reexpressed data, or the data sink can dynamically apply the inverse reexpression function to reexpressed data received at runtime. Like with the data source, static transformation of the data sink should always be chosen, if possible, because it will protect the data sinks from Type IV vulnerabilities. However, if the data sink needs to recover the original non-reexpressed data (e.g., to use as output), then dynamic inverse reexpression must be applied.

The data sinks are constructed by replicating the original data sink and implementing the determined inverse reexpression mechanism for each variant. Dynamic inverse reexpression mechanisms can be implemented as functions in the data sink.

Static transformation is accomplished in the same way as in the data sources, by applying the appropriate reexpression function to all relevant data encoded in the data sink. Operations on the data may also be transformed to preserve the original semantics.

5.1.8 Step 8 – Define semantics of divergence among the variants

The system must define the semantics of a divergence so that the monitor can be created to check for divergences. Divergence is stated in terms of variant states and/or data produced by the variants. Cox et. al modeled the normal equivalence property of N-variant systems as a means for reasoning about variant states [16].

The semantics of divergence is determined by considering the data being protected. If the protected data can be validated at the data sinks, then a divergence occurs when any of the sinks receive invalid data. If the data can be corrupted successfully and correctly by causing the variant data sources or data sinks to enter different states, then a divergence occurs when the variants transition to different states. Several previous N-variant systems were concerned with this type of divergence and used system calls to synchronize and detect divergence in the states of the variants.

5.1.9 Step 9 – Create the monitor

The monitor is created to check for divergences defined in the previous step. The monitor must be created to receive appropriate data from the variants to determine if a divergence has occurred. It must also be created with the ability to prevent corrupted system output and apply recovery to the variants if it detects a divergence.

The monitor should be constructed with the desired recovery mechanism in mind. The desired recovery mechanism is dictated by the designers and system requirements.

For existing systems in which recovery was implemented as terminating the variants, the monitor was designed with the authority to terminate the necessary processes. For more complex recovery mechanisms, the monitor might need to be designed to store necessary information about the variants.

5.1.10 Step 10 – Create the input splitter

The first step in creating the input splitter is enumerating the possible sources from which the data sources can receive input. Once these sources are identified, the input splitter should be created so that input from all of these sources is intercepted and replicated to each data sink.

5.1.11 Step 11 – Create the output merger

The output merger is created in a manner similar to the input splitter. First, all entities (processes, machines, clients, etc.) that receive output from the variants should be enumerated. For each of these entities, it must be decided whether that entity expects only one output from the system. If it does, then the output merger must be created so that it intercepts all variant outputs intended for that entity and combines them into a single output.

Using the address space partitioning example, the machine hardware and server clients are both external entities that receive output from the system. The variant web servers run as separate processes on the machine hardware, and so output intended for the machine, such as instructions, do not need to be merged. Server clients, however, expect that they are connected to a single server. Output intended for any clients is thus merged before being sent out.

5.1.12 Step 12 – Verify functional correctness

The N-variant version of a system must provide the same functionality as its original non-N-variant counterpart. To determine this, the data sources and data sinks can be analyzed to show that they transition through the same states as their original counterparts on all inputs. They also must produce outputs equivalent to those produced by their original counterparts for the same set of inputs.

Another way to test for functional correctness is to execute both the original and N-variant systems with identical sets of input. The effects should be the same for both systems (for non-malicious inputs). This technique is useful for testing for functional correctness on common types of input.

Chapter 6: Analysis of N-Variant Systems

The N-variant model enables a detailed analysis of existing N-variant systems. This chapter evaluates how well the model enables these analyses by examining five existing N-variant systems. An analysis of each system is presented using the model as a guiding framework. Each analysis identifies the system's N-variant components and properties predicted by those components. Additionally, each system is analyzed in terms of the four vulnerability types, using the model to show the flaws. Each section in this chapter discusses one of the five analyzed N-variant systems.

The general model provides the generic components and structure that N-variant systems have. We hypothesize that all N-variant systems are derivable from this general model. As part of the evaluation of the general model, an instantiation of the model is shown for each system as it was derived from the general model. The instantiations show how each system fits the general model. Each section explains how parts of a system's implementation map to each N-variant component. Grayed out parts of a system diagram indicate components of the general model that are not present in that system.

6.1 Analysis of Address Space Partitioning

Vulnerabilities within applications commonly exist that allow an attacker to overwrite memory contents with absolute memory address values in a process's address

space [16]. These vulnerabilities include format string [37], integer overflow, and double-free [17]. Cox et. al demonstrated how N-variant systems could be used to thwart these types of attacks using disjoint address spaces [16].

The security goal of this system was ensuring that memory addresses overwritten with absolute values are not accessed by the machine. An attacker who can insert absolute addresses into an address space can exploit this opportunity to gain control of a process. In order to prevent attacks that accessed specific absolute addresses, a 2-variant system was created in which the addresses of the two variants were reexpressed.

Figure 7. Address Space Partitioning Source and Sink

Figure 7 shows the data source and data sink of the system. The target data is addresses. The Apache web server [2] is the source that produces addresses and the hardware memory manager processes the addresses.

An instantiation of the general N-variant model for the address space partitioning system is shown in Figure 8.

6.1.1 Data Sources

The Apache web server represents the data source for this system. Two data sources were created to operate in two different address space variations. Reexpression functions were used to transform memory addresses in each data source to a respective reexpressed form. The reexpression function used by the first data source was the identity function, $R`(a) = a$. The reexpression function used by the second data source

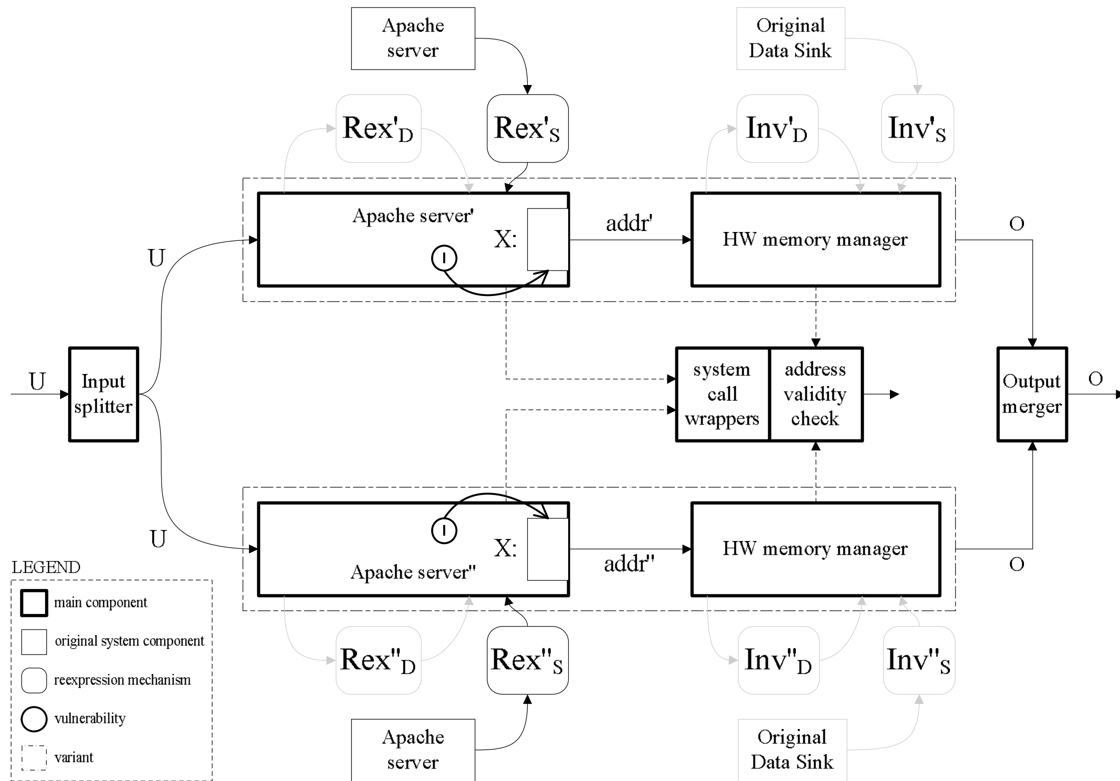

Figure 8. Address Space Partitioning Model

was $R``(a) = a + 0x80000000$. This second function effectively changed the high order bit of every address to a 1. These two functions satisfy the disjointedness property required by the model because the first function produces addresses whose high bit is 0, while the second function produces addresses whose high bit is 1 (for programs which use only 31-bits of address space).

All addresses within a data source must be transformed using the respective reexpression function. The general model indicates that this can be done dynamically or statically. This system used static reexpression.

The static reexpression was a transformation of the data source (web server) binary. The source code for both data sources remained identical. A linker script was used to create the two data sources. The linker script applied the reexpression functions

indirectly by loading the code and data segments of each data source at their respective starting addresses, according to the reexpression functions. This effectively created an address space for each data source in which the addresses used were only those within the regions defined by the reexpression functions. Additionally, a limit was placed on the size of the first source's data segment, so that it could not grow into the second source's address space.

6.1.2 Data Sinks

The data sink in this system was the hardware memory management, which interprets the addresses of a data source. The data sinks were not explicitly created for this 2-variant system. Instead, they were given as part of the machine and were aware of the memory regions of each data source because of the mechanisms of memory management. As a consequence of this, the data sinks did not need to apply inverse reexpression. They operated on reexpressed data by knowing the address space layouts of the data sources. The inverse reexpressions were thus implicit in the functions of the data sinks. These implicit inverse reexpression functions were $R`^{-1}(a) = a$ for the first data source and $R``^{-1}(a) = a - 0x80000000$ for the second data source.

6.1.3 Input Splitter

The operating system performed the input splitting in this system. Wrappers were implemented around system calls that performed input for the data sources. An actual input operation was performed only once, but the wrappers ensured that the same input data was sent to all data sources. This input splitter construction satisfies the necessary properties of the input splitter.

6.1.4 Monitor

The monitor in this system was the hardware memory management and the system call wrappers. It monitored for divergence in two ways. The first way of monitoring directly detected an injected address. Because of the disjoint memory regions of the data sources, an injected address is guaranteed to be invalid in one of the two data sinks. The hardware memory management facilities detect this and produce a memory access error. The second way of monitoring detected a divergence in the behaviors of the data sources. Wrappers implemented around operating system call procedures checked for N-variant processes and ensured that the variant data sources made the same system calls with equivalent arguments.

When the monitor detected a divergence, it terminated both data source processes. In the case of a memory access error, the compromised data source had already terminated by crashing. Terminating the data sources put the two variants back into an uncompromised (though also useless) state. Divergence was always detected by a wrapper before the main system call procedure executed. This prevented a compromised system from corrupting any external state.

6.1.5 Output Merger

Output from the variants was merged using the system call wrappers. The wrappers ensured that each variant made the same output system call with equivalent parameters, and then performed the actual output operation only once. Addresses were not merged after being processed by the data sinks because the two variants were designed to run as two separate processes with distinct memory regions on the machine.

6.1.6 Vulnerability Analysis

Type I: The reexpression functions in this system modified only the highest bit of addresses to create disjoint memory regions for each of the data sources. An attacker who can overwrite an existing address without overwriting the highest bit can construct addresses that are valid in both data sources. This represents a Type I vulnerability if an attacker can partially overwrite memory addresses. The general model requires that an N-variant system susceptible to partial overwrites provide claims about why they are not a security risk. In the paper, the authors acknowledged the vulnerability, but relied on the assumption that the attacker must construct the entire address. They admit that this assumption might not be valid for all scenarios. Thus the system is considered susceptible to Type I vulnerabilities in the form of partial memory overwrites. Bruschi et. al addressed these partial memory overwrite vulnerabilities [9], and their system is analyzed in the next section.

Type II: The data source reexpression mechanism was a static transformation of addresses, performed by a linker script. Absolute addresses are not reexpressed dynamically in the data source, and thus the system is immune to Type II vulnerabilities.

An attacker could construct addresses that are valid in both data sources by corrupting values that are used with reexpressed addresses in address calculations. This represents an attack that relies on corrupting relative memory locations. The desired security property of this system was ensuring memory addresses overwritten with *absolute* values are not accessed by the machine. Relative address corruptions are thus considered out of the attack scope and do not represent a Type II vulnerability.

Type III: This system relied on operating system mechanisms to provide data source isolation. The data sources execute as separate processes and are therefore isolated from other components. The memory protection provided by the operating system prevents vulnerabilities in the data sources (web servers) from directly affecting other system components.

Type IV: The inverse reexpression mechanism is completely static because the data sinks are able to process address data in reexpressed form. Thus the system is immune to Type IV vulnerabilities.

6.2 Analysis of Enhanced Address Space Partitioning

The address space partitioning system introduced by Cox et. al (analyzed in Section 6.1) was susceptible to Type I vulnerabilities in the form of partial memory overwrites. Bruschi et. al addressed these vulnerabilities by enhancing the address space partitioning system to thwart partial memory overwrites in addition to complete address overwrites [9].

The security goal of this system was ensuring that memory addresses wholly overwritten with absolute values *and partially overwritten with absolute values* are not accessed by the machine. An attacker who can corrupt the least significant bytes of a memory address can execute an Impossible Path Execution (IPE) attack [43]. This type of attack can force the exploited program to bypass security-critical code.

The enhanced address space partitioning system was based on the same framework used in the original address space partitioning system. Most of the system components are the same, and the models are identical (Figure 8 in Section 6.1). Only

the parts of components that were altered due to the enhancement are explained in this section. These components include the data sources, data sinks, and monitor. Additionally, an updated analysis of vulnerabilities is provided.

6.2.1 Data Sources

This system used the same data sources as the original address space partitioning system, but with different reexpression functions. The reexpression function used by the first data source remained the identity function, $R`(a) = a$. The reexpression function used by the second data source was $R``(a) = a + 0x80000000 + k$. The k represented a fixed number of bytes that the memory segments of the second data source were shifted. Shifting the memory segments of the second data source diversified relative distances between the two source address spaces. Logically equivalent memory contents now existed at different addresses.

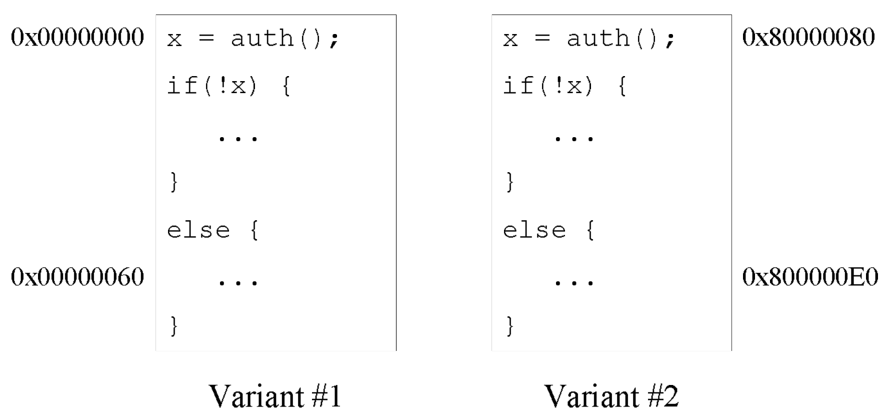

Figure 9. Address Shifting Example

Figure 9 shows a simplified example in which addresses have been shifted by 16 bytes in the second variant. An attacker could overwrite the lower byte in the return address of *auth()* with *0x60* in order to branch execution of the first variant to the else

clause and bypass the security mechanism. However, this attack would cause the second variant to branch somewhere before this code, thus resulting in unexpected and divergent behavior from the first variant.

The data sources applied the reexpression functions statically with a linker script as in the original system. To achieve the offset, "junk" data was inserted to cause a shift in the second data source's addresses. Additionally, the handling of dynamically-linked binaries had to be modified to perform the same shifting.

6.2.2 Data Sinks

The hardware memory management served as the data sink, and as a result, the data sinks did not need to apply inverse reexpression. They operated on reexpressed data by knowing the address space layouts of the data sources. The inverse reexpressions were thus implicit in the functions of the data sinks. These implicit inverse reexpression functions were $R`^{-1}(a) = a$ for the first data source and $R``^{-1}(a) = a - 0x80000000 - k$ for the second data source, where k was the fixed shift amount.

6.2.3 Monitor

The monitor remained the same as it was in the original address space partitioning system. Partial memory overwrites were detected by system call equivalence monitoring. As Figure 9 showed, a partial memory overwrite would corrupt addresses in logically different ways and cause the behaviors of the two data sources to diverge. This would *probabilistically* lead to divergent system calls being made by the data sources, which would be detected by the monitor. Detection is only probabilistic because there is no guarantee that the divergent behaviors will lead to divergent sequences of system calls.

Consider the example from Figure 9 again. If the else clause was large enough, an attacker could partially overwrite the return address of *auth()* to branch both variants past the security check. If no system calls were made between the branch destinations, then a divergence would not be detected. This probabilistic property of detection affects Type I vulnerability security claims and is explained in the next section.

6.2.4 Vulnerability Analysis

Type I: The original address space partitioning system was susceptible to Type I vulnerabilities in the form of partial memory overwrites. This enhanced version attempted to eliminate these vulnerabilities, but did so only probabilistically. Partial memory overwrite attacks are detected by the monitor as a divergence in the system calls made by the data sources. However, the granularity of system calls is not fine-grained enough to guarantee that divergent behavior will lead to divergent system calls. It is highly likely that any meaningful attack will be detected, but the implementation of the monitor does not allow a guarantee to be made against partial memory overwrites.

Type II: The reexpression of dynamically linked binaries seems to create Type II vulnerabilities because the address shifting is performed at runtime. However, exploiting this vulnerability would require an attacker to dynamically link his own binary to the executable. This type of attack would have more serious implications than partial memory overwrites and is considered out of the scope of this system.

Type III and Type IV: These vulnerabilities remain the same as they were in the original address space partitioning system.

6.3 Analysis of Instruction Set Tagging

Cox et. al used the same framework from the address space partitioning system to demonstrate an N-variant system that could thwart code injection attacks [16]. The security goal of this system was to prevent unintended code from executing on the target machine. In order to prevent attacks that inject code, a 2-variant system was created in which reexpressed tags were added to instructions.

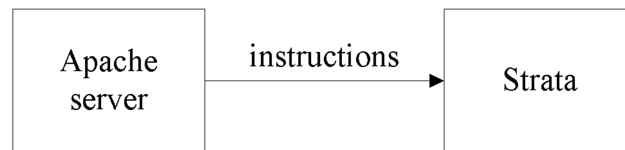

Figure 10. Instruction Set Tagging Source and Sink

Figure 10 shows the data source and data sink of the system. The target data is instructions. The Apache web server is the source that produces instructions and the Strata virtual machine processes the instructions.

An instantiation of the general N-variant model for the instruction set tagging system is shown in Figure 11.

6.3.1 Data Sources

This system used the Apache web server as the data source[2]. Instructions in each data source were transformed by prepending appropriately reexpressed tags. The reexpression function used by the first data source was $R`(inst) = 10101010 \| inst$. This function prepended the byte-long tag '10101010' to each instruction. The reexpression function used by the second data source was $R``(inst) = 01010101 \| inst$. This function prepended the byte-long tag '01010101' to each instruction. This system did not use the identity function for either of the reexpression functions because that would not have

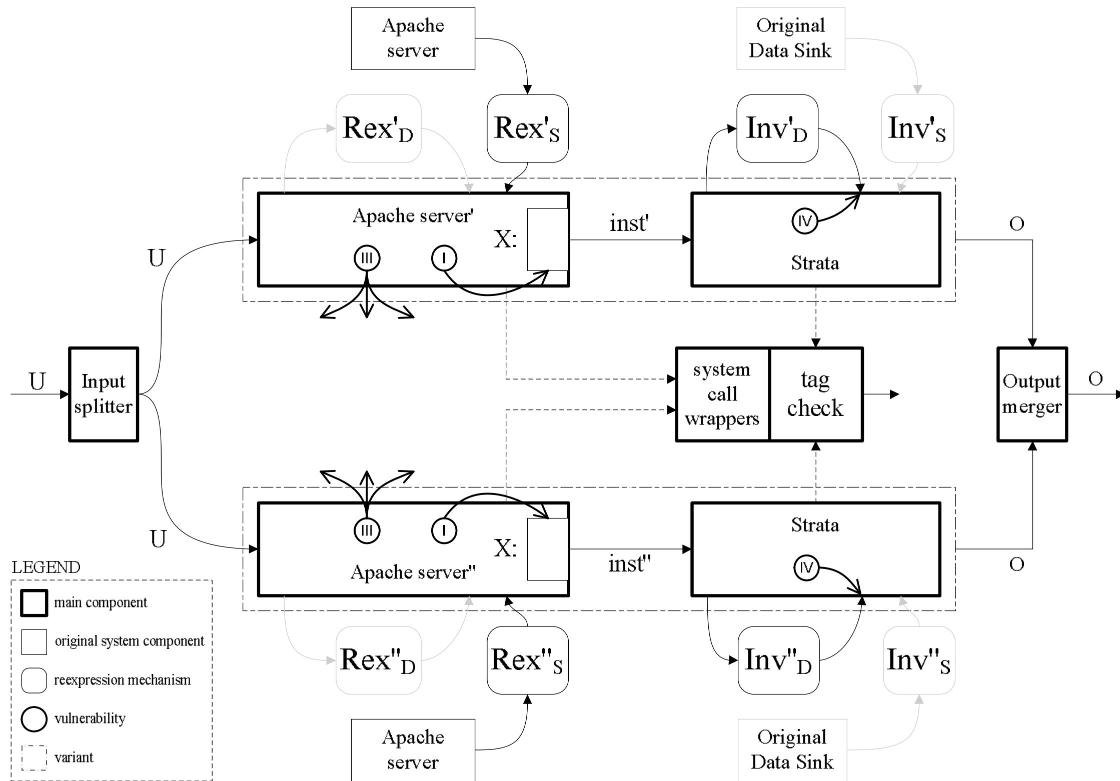

Figure 11. Instruction Set Tagging Model

added any tags to the instructions. The two reexpression functions satisfy the disjointedness property because the tags are clearly different.

All instructions within a data source must be transformed using the respective reexpression function. The general model indicates that this can be done dynamically or statically. This system used static reexpression.

The static reexpression was a transformation of the data source (web server) binary. Diablo, a static binary rewriting framework [40], was used to rewrite the data source binaries to insert the tags. This binary rewriting was applied before execution and therefore represents a completely static transformation.

6.3.2 Data Sinks

The data sink in this system was Strata, a software dynamic translation tool [35]. Strata is a VM that examines and translates instructions before they execute on the host machine. Strata was used by this system to check and remove instruction tags before they reached the processor. By removing the tags, Strata applied inverse reexpression functions to the data source instructions. The inverse reexpression function applied by the first Strata data sink was $R^{`-1}(10101010 \,||\, inst) = inst$. The inverse reexpression function applied by the second Strata data sink was $R^{``-1}(01010101 \,||\, inst) = inst$. Both of these functions are obvious inverses because they only remove the tag that was added during reexpression.

Strata applied the inverse reexpression functions dynamically during execution. The general model indicates that any dynamic inverse reexpression mechanism is potentially susceptible to Type IV vulnerabilities. This is addressed in the vulnerabilities discussion later in this section (Section 6.3.6).

6.3.3 Input Splitter

The input splitter was implemented in the same way as in the address space partitioning system. The operating system performed the input splitting through wrappers around input system calls. An input operation was performed only once, and the received input was sent to all data sources.

6.3.4 Monitor

There were two monitors in this system, each monitoring for a divergence in a different way. The first monitor was the Strata VM. Strata's instruction fetch module

was modified to check that each fetched instruction had the correct tag for its variant. If a tag was not correct, then a security violation was detected. The second monitor was implemented in the system call wrappers. This second monitor used the same system call observing that was used by the address space partitioning system to ensure that each data source made the same system call with the same parameters.

When either of the two monitors detected a divergence, the data sources were terminated. This put the two variants back into an uncompromised state. If Strata detected the divergence in the form of an incorrect tag, then the associated instruction was not executed. This prevented corrupted code from reaching the processor. A divergence observed by the system call wrappers was always detected before the system call procedure executed. This prevented a compromised system from corrupting any external state.

6.3.5 Output Merger

The output merging was implemented in the same way as in the address space partitioning system. Output from the variants was merged using the system call wrappers. Instructions were not merged after being processed by Strata because the two variants were designed to run as two separate processes on the machine.

6.3.6 Vulnerability Analysis

Type I: The reexpression functions used in this system prepended reexpressed tags to instructions. They did not reexpress the rest of an instruction. If memory was bit-addressable, then an attacker could overwrite the part of the instruction after the tag while still preserving the tag. This represents a Type I vulnerability if an attacker can partially

overwrite instructions. The general model requires that an N-variant system susceptible to partial overwrites provide claims about why they are not a security risk. The authors of the paper explained that for all known realistic code injection attacks, partially overwritten instructions were not considered a serious risk [16].

Type II: Data source instructions were rewritten statically before execution, and so the reexpression mechanism was not vulnerable to Type II vulnerabilities. However, a data source allowed to modify its own instructions could be susceptible to Type II vulnerabilities. If an attacker could inject code into memory, and then use existing code to transform the injected memory differently in each data source, then the attacker could conceivably inject code that would be transformed differently by each data source. This would represent a Type II vulnerability. The authors of the paper addressed this type of vulnerability and explained that it was not a realistic threat.

Type III: The Strata VM is not strongly isolated from the data sources. However, Strata provides protection against Type III vulnerabilities by protecting its critical data structures. The authors explain that "before switching to the application code, the Strata VM uses `mprotect` to protect critical data structures including the fragment cache from being overwritten by the application" [16]. The Strata VM also was subject to two red team evaluations that were unable to find any major VM implementation flaws [45]. Additionally, the memory protection capabilities of the operating system prevented Type III vulnerabilities in one data source from corrupting the other data source.

Type IV: The data sinks (Strata) applied the inverse reexpression functions at runtime in order to remove the instruction tags. Strata's instruction fetch module was modified to check an instruction for the correct tag, remove the tag if it was correct, and

then place the actual instruction in the fragment cache where it is executed directly on the host CPU. A data sink would contain Type IV vulnerabilities if an attacker could overwrite instructions in the fragment cache. However, Strata uses `mprotect` to protect the fragment cache from being overwritten.

6.4 Analysis of UID Variations

Chen et. al demonstrated the viability of non-control data attacks [13]. These attacks overwrite security critical data, such as user identification values, configuration data, and decision making data, without altering the control flow of the exploited program. Nguyen-Tuong et. al demonstrated how N-variant systems could be used to thwart such attacks [29], using user identification (UID) data to demonstrate.

The security goal of this N-variant system was to prevent corrupted UID data values from being processed by the operating system. If an attacker can corrupt UID data, then he can raise the privileges of the exploited program and use it to masquerade as any user in a system, particularly the *root* user on a Unix system. In order to prevent attacks that rely on corrupting UID values, a 2-variant system was created in which all UID values were reexpressed by the variants.

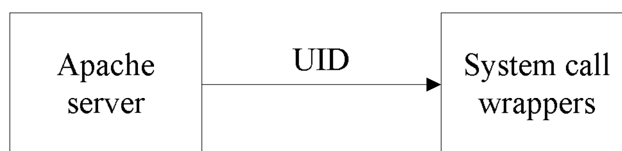

Figure 12. UID Variations Source and Sink

Figure 12 shows the data source and data sink of the system. The target data is UID values. The Apache web server is the source that produces UID values and system call wrappers are built to process the UID values.

An instantiation of the general N-variant model for the UID variations system is shown in Figure 13.

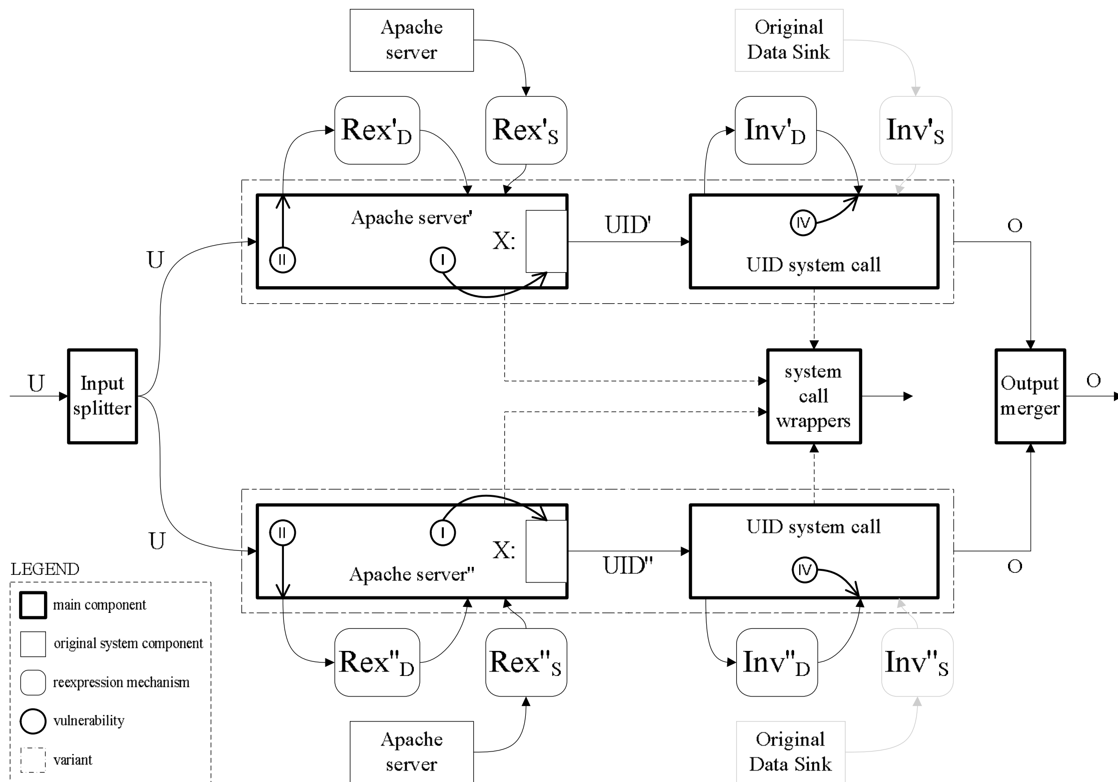

Figure 13. UID Variations Model

6.4.1 Data Sources

The Apache web server was chosen as the data source for this system [2]. Two data sources were created to operate with two UID variations. They used reexpression functions to transform all UID data within their programs to a respective reexpressed form. The reexpression function used by the first data source was the identity function, $R`(u) = u$. The reexpression function used by the second data source was the XOR function, $R``(u) = u \oplus 0x7FFFFFFF$. This function uses XOR to flip bits and was chosen so that the result would be a bit pattern that differed in all positions except the most significant bit. Flipping bits will always change the value, and so the two functions

satisfy the disjointedness property required by the model. Not flipping the high bit leaves the system potentially susceptible to Type I vulnerabilities. This is addressed and discussed in Section 6.4.6.

All UID data within a data source must be transformed using the respective reexpression function. The general model indicates that this can be done dynamically or statically. This system used a combination of both dynamic and static reexpression.

The static reexpression was a transformation of UID values in the server's source code. The first data source applied the identity reexpression function and used the original server program unchanged. For the second data source, all UID values in the server source code were transformed using the second reexpression function.

The dynamic reexpression in the data sources was not explicitly recognized in the original design of the system. The dynamic aspect of the reexpression was located in the data source's support for external UID data. The original Apache web server relied on trusted external files to obtain some UID values. In the N-variant system, the data sources needed to receive differently reexpressed UID data from these external sources. Applying the reexpression functions as data was read in from these external sources was ruled out because an attacker could corrupt data by using the same path. The solution settled on was to create two versions of each external file and apply reexpression to their UID contents. Each data source would then read appropriately reexpressed UID data from its own version of the file.

This solution seemed to remove dynamic reexpression from the data sources because UID values were statically modified to their reexpressed forms in the two external file versions. However, there was still a dynamic aspect to this approach. The

two files mapped user names to user identification values. Obtaining a UID value from a file required a dynamic lookup based on the user name. Figure 14 illustrates the lookup process graphically for both data sources. Reexpressed UID data is acquired at runtime based on dynamically generated input to the external file, and thus represents a form of dynamic reexpression. A dynamic reexpression mechanism is potentially susceptible to Type II vulnerabilities. This is addressed and discussed in Section 6.4.6.

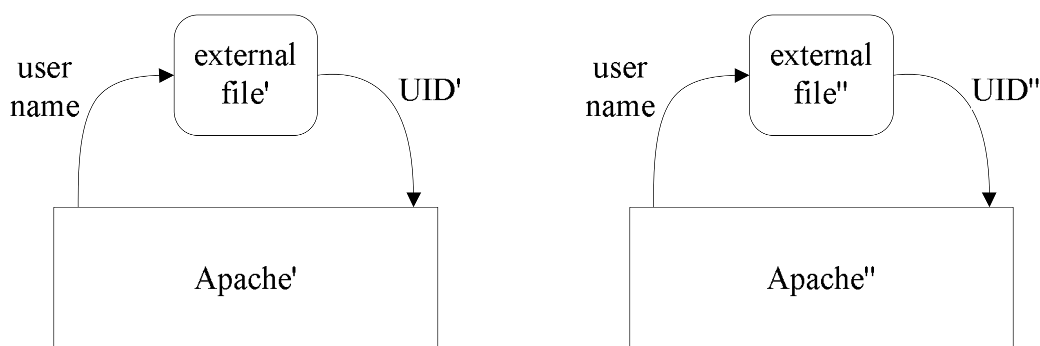

Figure 14. Dynamic Reexpression in UID System Data Sinks

6.4.2 Data Sinks

The goal of an attack on this system was to get corrupted UID data to the operating system in order to drop or escalate privileges. Thus, the system call wrappers represented the data sinks for this system. The wrappers applied inverse reexpression functions in order to regain original UID values for correct operation. The first variant's inverse reexpression function was $R`^{-1}(u) = u.$ The second variant's inverse reexpression function was $R``^{-1}(u) = u \oplus 0x7FFFFFFF$.

The code that processed system calls had to be made to operate with reexpressed data. The general model indicates that this can be done either dynamically by reversing

the reexpressions or statically by constructing the data sinks to operate on the reexpressed data. This system applies the inverse reexpressions dynamically.

The kernel was modified to keep track of the variant web servers and to implement wrappers around the system calls. The wrappers of all system calls that take UID parameters were modified to apply the appropriate inverse reexpression function to the parameters at the beginning of the procedure. This dynamic inverse reexpression is potentially susceptible to Type IV vulnerabilities. This is addressed and discussed in Section 6.4.6.

6.4.3 Input Splitter

The input splitter was implemented in the same way as in the address space partitioning and instruction set tagging systems. Input operations were performed once, and system call wrappers ensured that the same input was sent to all data sources.

6.4.4 Monitor

The monitor in this system was the system call wrappers. They monitored for a divergence in two ways. The first way of monitoring directly detected corrupt UID values. After applying the inverse reexpression to UID parameters, the UID system call wrappers checked that the same post-inverse reexpression values were passed to the system call by both data sources (web servers). The second way of monitoring detected a divergence in the execution sequences of the servers. This was accomplished by observing the system call sequences of the servers and ensuring that they were equivalent.

When the monitor detected either of the two divergence types, it terminated both web server processes. This put the two variants back into an uncompromised state. The monitor also prevented the system call code that detected the divergence from executing its function. This prevented the operating system from using corrupted data.

6.4.5 Output Merger

Like the input splitting, the output merging was performed by the operating system. For UID and other output related system calls, the wrappers checked that the variants made equivalent system calls with equivalent parameters. If they did, then the actual system call operation was performed only once.

6.4.6 Vulnerability Analysis

Type I: The reexpression functions for the two variants were chosen so as to produce UID values that differ in all positions except the most significant bit. The result of this choice is that the reexpressions are susceptible to a high-bit overwrite, since the high bit is not reexpressed. This represents a Type I vulnerability if UID values can be partially overwritten. The general model requires that an N-variant system either use reexpression functions that are not vulnerable to partial overwrites or provide claims about why they are not a security risk. In the paper, the authors explain that "although individual bit overwrites are certainly possible in theory, the lowest level of granularity reported for partial memory overwriting attacks under a remote attacker threat model is at the byte-level so we do not consider this a likely threat" [29]. Theoretically, the system remains vulnerable to a Type I vulnerability in the form of a high bit overwrite, but realistically, the vulnerability is not a threat.

Type II: Section 6.4.1 explained how the data sources obtained reexpressed UID values from external files using a dynamic lookup based on user name. The model indicates that any dynamic reexpression mechanism is susceptible to Type II vulnerabilities. In this case, an attacker could overwrite the input to the lookup function (a string), and the results would be correctly reexpressed UID values for both data sources. This kind of attack would allow an attacker to indirectly corrupt UID values. The identification of Type II vulnerabilities here shows how they can exist even in systems that appear at first to use only static reexpression.

Type III: The system provides protection against Type III vulnerabilities because of the isolation of the data sources. The data sources (web servers) are implemented as processes, and therefore execute in separate memory spaces from other components. This isolation prevents vulnerabilities in the web server from directly affecting these other system components.

Type IV: Section 6.4.2 explained how the data sinks applied inverse reexpression functions at runtime to obtain original UID values. The general model indicates that any dynamic inverse reexpression mechanism is potentially susceptible to Type IV vulnerabilities. In this case, an attacker could overwrite the outputs of the dynamic inverse reexpressions with UID values. This would completely bypass the reexpression mechanisms and allow an attacker to get corrupted UID data into the data sinks. However, the inverse reexpression mechanisms are very simple (applying identity function and XOR function), and strong arguments can be made regarding their security.

6.5 Analysis of Stack Reversal

M. Franz noted that malicious insiders working within commercial software companies could purposefully insert vulnerabilities into an application [20]. To eliminate the malicious insider advantage, Franz proposed an N-variant system in which the variants used different stack layouts. Salamat et. al demonstrated a 2-variant system called Orchestra in which the stacks of the two variants grew in opposite directions [33].

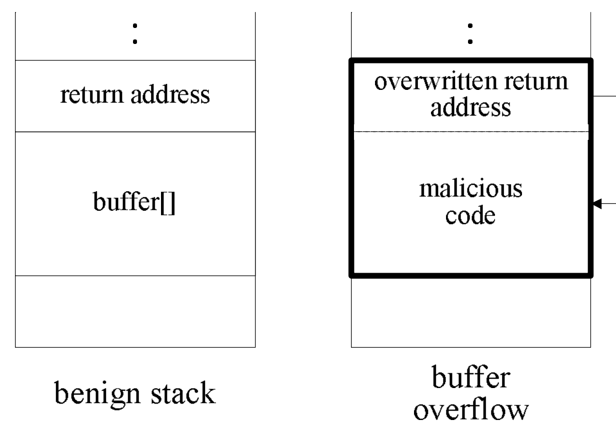

Figure 15. Buffer Overflow Attack

The security goal of this system was to prevent attacks that rely on knowledge of the stack layout. An attacker can use knowledge of the stack layout to devise an attack that targets data on the stack. For example, Figure 15 shows an attack which exploits a buffer overflow vulnerability to overwrite a return address and branch to the buffer which was filled with malicious code.

Figure 16 shows the data source and data sink of the system. The target data is the contents of the stack. The dashed box indicates a program in which both the data source and data sink reside. The data source is the part of this program that produces stack activations records while the data sink is the part that consumes them.

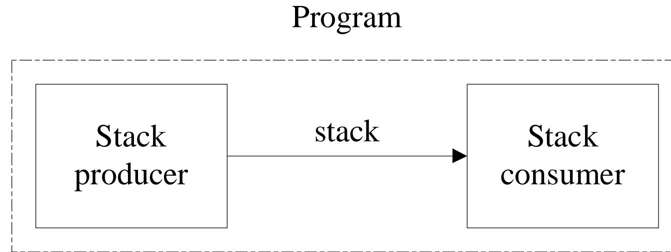

Figure 16. Stack Reversal Source and Sink

An instantiation of the general N-variant model for the stack reversal system is shown in Figure 17.

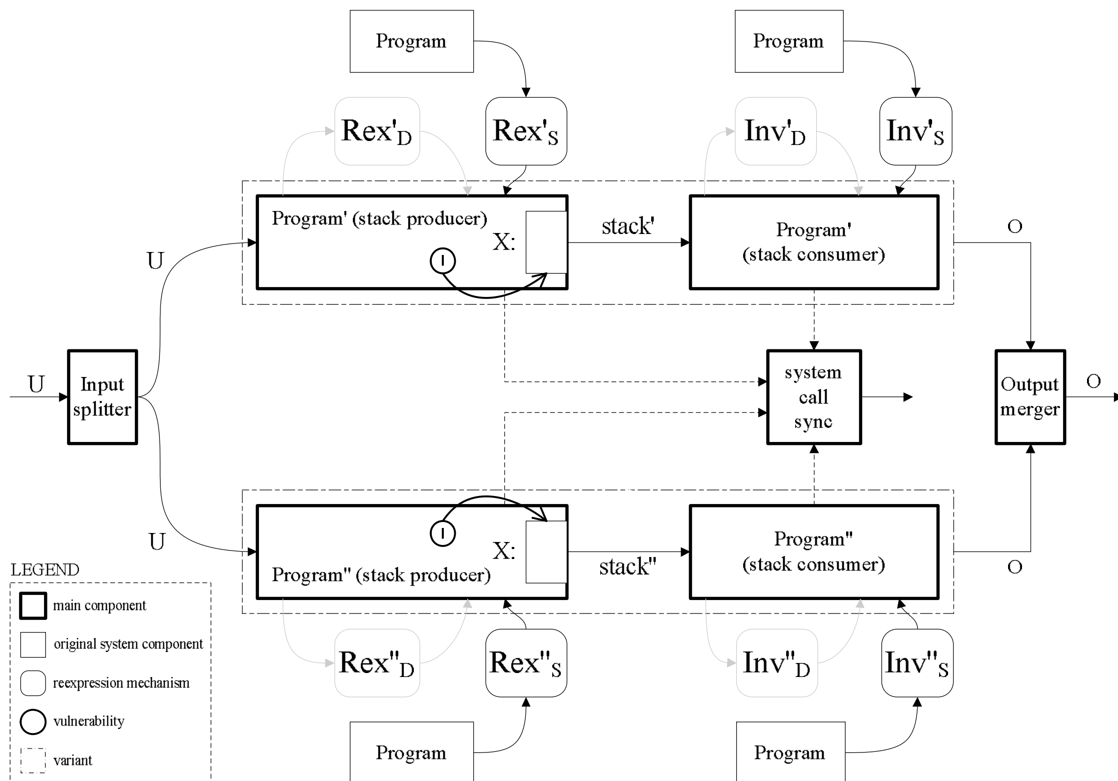

Figure 17. Stack Reversal Model

The stack reversal system was implemented using five different original programs for evaluation purposes. These programs were *find*, *tar*, an MD5 sum generation program, *apache*, and *SPEC CPU2000*. The data source and data sink for each system were both part of this original program. The data source consisted of the parts of the

program that produced stack activation records. The data sink consisted of the parts of the program that consumed stack activation records.

6.5.1 Data Sources

In the N-variant system, two data sources produced two variations of the stack. The reexpression function used to create the first data source was the identity function, $R`(stack) = stack$. The reexpression function used to create the second data source was $R``(stack) = reverse(stack)$. The first data source wrote the stack conventionally (downward on x86), while the *reverse(stack)* function indicates that the second data source reversed the direction of stack growth.

6.5.2 Data Sinks

The two data sinks consumed the two reexpressed variations of the stack. The data sinks were statically reexpressed by a compiler to know the directions of stack growth. The first inverse reexpression function was implicitly $R`^{-1}(stack) = stack$. The second inverse reexpression function was implicitly $R``^{-1}(stack) = reverse(stack)$.

Both the data sources and data sinks used static reexpression mechanisms. The static reexpression was a transformation of the program (which included the data sources and data sinks) binary. A normal compiler was used to create the first variant program binary because the conventional stack direction was used. The Orchestra compiler was used to produce the second variant program binary in which the direction of the stack was reversed.

6.5.3 Input Splitter

This system used a separate user-space process to implement the functionalities of the input splitter, output merger, and monitor. Salamat et. al called this process the Monitor, but it performed the functions of all three of these N-variant components. Throughout the remainder of this section, this process will be referred to as the Monitor (capital M), and the N-variant monitoring component will be referred to as the monitor (lower case m).

The Monitor performed the input splitting for the stack reversal system. It operated similarly to the input splitters from the address space partitioning and instruction set tagging systems, but it was implemented in user-space instead of in the operating system. The Monitor created the two variants as child processes and had the ability to intercept their system calls. Whenever the variants issued input system calls, the Monitor would check the equivalence of the calls, suspend the two variants, intercept the input, and then send identical copies of the input to both variants.

6.5.4 Monitor

The Monitor process was also responsible for divergence detection. It monitored for a divergence by synchronizing the system calls made by the two variant processes. When one of the variants issued a system call, the Monitor intercepted the request and suspended the variant. The Monitor then tried to synchronize the system call with the other variant by checking that both variants made the same system call with equivalent parameters within a certain window of time.

If the Monitor detected a divergence in the system calls, their parameters, or their issue times, then the Monitor raised an alarm, terminated the variants, and then restarted

the variants. Terminating and restarting the variants put them into an uncompromised state while still allowing them to provide service again.

6.5.5 Output Merger

The two variants were designed to be seen as two separate processes by the machine, so stack representations were not merged. However, other outputs of the variants were merged by the Monitor. When the variants tried to output data, the Monitor suspended the variants, intercepted the output, checked for equivalent output data, and then performed the output operation only once.

6.5.6 Vulnerability Analysis

Type I: The stack reversal system probabilistically mitigated Type I vulnerabilities. The intuitive argument against Type I vulnerabilities in this system was that a corruption of the stack would quickly result in divergent behavior of the two variants. However, divergence monitoring occurred only at the granularity of system calls. It is possible for an attacker to execute an attack that would not lead to divergent system calls. The authors acknowledge this vulnerability in the Orchestra paper but note that this is a high barrier for an attacker to overcome. Still, this type of attack is difficult but not impossible, and the authors suggest that complementary defenses should be used for high security applications.

Type II: This system should be immune to Type II vulnerabilities because of the static reexpression mechanism performed by the Orchestra compiler. However, if there existed logic in the variants that inspected the state of the program and made decisions based on the direction of the stack, then this would indicate a vulnerability that could be

exploited to carry out an attack. An analysis of the code generated by the compiler could confirm the absence of this type of program logic, and therefore the absence of Type II vulnerabilities.

Type III: The Orchestra architecture explicitly addressed Type III vulnerabilities. The Monitor and the two variants were implemented as separate processes with their own address spaces. The defenses against Type III vulnerabilities relied on the memory protection capabilities provided by the operating system. The isolation of the processes meant that none of these components could directly manipulate the memory space of another component. It was therefore difficult to compromise the Monitor or another variant by taking control of one of the variants.

Type IV: This system should be immune to Type IV vulnerabilities because of the static transformation applied by the Orchestra compiler. Like Type II vulnerabilities, this could be confirmed by an analysis of the code generated by the Orchestra compiler.

6.6 Summary of Vulnerabilities

	Address Space Partitioning	Enhanced Address Space Partitioning	Instruction Set Tagging	UID Variations	Orchestra - Stack Reversal
Type I	Partial	Probabilistic	Partial	Partial	Probabilistic
Type II	X	X	X	V	X
Type III	X	X	V	X	X
Type IV	X	X	V	V	X

Table 1. Vulnerabilities in Existing N-Variant Systems

A summary of the vulnerabilities in each of the five analyzed systems is given in Table 1. This table indicates what vulnerabilities exist in each of the five systems. In the

table, a ‘V’ indicates that the system is vulnerable to that vulnerability type, though these vulnerabilities might have been mitigated in the system by some means of protection. An ‘X’ indicates that the system is not susceptible to that vulnerability type. A ‘Probabilistic’ indicates that the system probabilistically protects against that vulnerability type but might be at risk under certain conditions. A ‘Partial’ indicates a vulnerability to partial overwrites.

Analyzing these five N-variant systems with regards to the new model has yielded useful insights into the four vulnerability types:

- **Type I:** N-variant systems using disjoint data reexpression functions can make fundamentally stronger claims with respect to preventing Type I vulnerabilities compared to systems not using disjoint data reexpression functions. This is because the disjointedness property between the two functions guarantees that an attacker cannot construct complete attack values that will be valid in all variants. However, as the above analyses have shown, Type I vulnerabilities can still exist in systems that offer only probabilistic protection and in systems where the reexpressed data are vulnerable to partial overwrites. Analysis of a system would reveal these types of vulnerabilities and, if present, they must be addressed.
- **Type II:** Data sources that use any type of dynamic reexpression mechanism are susceptible to Type II vulnerabilities, while only data sources that use completely static reexpression mechanisms are immune to them. However, as the analysis of the UID variations system showed, Type II vulnerabilities can exist even in systems that at first appear to be immune. For systems that use dynamic

reexpression, the Type II vulnerabilities must be identified and the risks of these vulnerabilities carefully evaluated.

- **Type III:** These vulnerabilities can be mitigated by providing isolation between components of the system. This has been achieved in previous systems mainly by relying on the memory protection facilities provided by the operating system. When components cannot be strongly isolated, other measures should be taken to protect the components from Type III vulnerabilities.
- **Type IV:** Data sinks that apply inverse reexpression functions at runtime are susceptible to Type IV vulnerabilities, while only data sinks that are statically transformed to operate with reexpressed data are immune to them. Systems that apply inverse reexpression functions at runtime must either protect against Type IV vulnerabilities or show that they are not a realistic threat. The instruction set tagging system demonstrated an example of defending against Type IV vulnerabilities by protecting Strata's fragment cache before control is switched back to application code.

Chapter 7: SQL N-Variant System

This chapter evaluates the N-variant construction process by presenting an experiment in designing and implementing a new N-variant system. This experiment demonstrates the use of the N-variant construction process outlined in Chapter 5 by following the provided synthesis steps. The general N-variant model and the construction process serve as guiding frameworks for construction. If the model and the construction process are correct, then we should be able to use them to correctly design and implement our new system.

This experiment will construct a 2-variant system that protects information stored in a database. The results of the experiment should produce a system that provides the desired security property without relying on secrets. We evaluate the resulting system according to:

- An assessment of the security provided.
- Its consideration of the 4 vulnerability types.
- A comparison to similar techniques.

7.1 System Model

The system model examined in this example is a web application that interfaces with external clients and a database. The overall architecture of the system is illustrated

in Figure 18. For the purposes of this example, the database stores sensitive credit card information. A client can access the database through the web application using specific queries that are set up by the application. The user provides inputs to the application that are used as parameters in these queries.

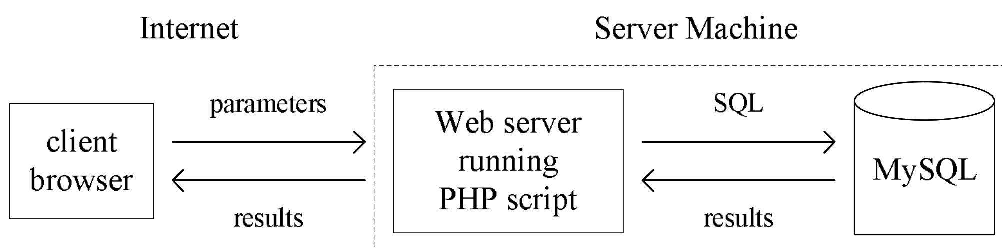

Figure 18. SQL System Original Architecture

The web application in this example is a PHP script. The database is a MySQL database that uses structured query language (SQL) for the retrieval of data. The PHP script has hard-coded SQL commands with placeholders for certain variables. The normal operation of the web application is to fill these variables with user input, create the SQL statement, and issue the SQL command to the database. The information returned by the database is then displayed back to the client.

7.2 Design Process

7.2.1 Step 1 – Identify desired security property

The system outlined in Section 7.1 interacts with a database. Because of its sensitive nature, data stored by the database should only be accessed in ways intended by the system. Thus, the desired security property of this system is:

Ensure that only intended operations are performed on the database.

An intended operation is an SQL statement that is provided by a trusted component of the system and whose structure is defined by that trusted component. The web application defines trusted SQL statement structures and issues queries having those structures to the database. Any queries received by the database from the web application having these structures are considered intended operations.

7.2.2 Step 2 – Identify information that needs to be protected

An attacker can gain unauthorized access to information in the database by issuing malicious SQL statements to the database. For example, an attacker can exploit vulnerabilities in the web application to perform an SQL injection attack [11]. An SQL injection attack changes the intended structure of an SQL statement [39]. In an SQL injection attack, the attacker's inputs are translated by the SQL interpreter as part of the structure of the query, instead of being confined to the parameters as intended. By carefully crafting the input to the web application, a malicious client can alter the intended SQL query issued by the web application or even inject his own SQL command. For example, consider a web application that constructs the following query:

SELECT acct FROM users WHERE login='$uid' AND pwd='$pwd'

The variables $uid and $pwd are supplied by the user. Under normal (non-attack) conditions, the user inputs are used as literals in the query, producing the following statement in which example user inputs are shown in bold:

SELECT acct FROM users WHERE login='**bob**' AND pwd='**foo**'

However, a malicious user can craft inputs so as to change the structure of the statement and produce a query with different intent:

SELECT acct FROM users WHERE login='**root' --** ' AND pwd='foo'

In this example, a malicious user has entered "root' --" for the $uid variable. The single quote escapes the literal, making root the effective login parameter. Normally, the $pwd variable would be checked as part of the query to validate the login. However, the remainder of the statement (shown in gray) is commented out by the "--" that was entered as part of the $uid parameter, effectively circumventing the password check.

Of course, SQL injection attacks are just one method by which an attacker can gain unauthorized access to information in the database. To achieve the desired security property, *the system cannot simply prevent a particular attack*. Instead, the system must ensure that only SQL statements having a structure defined by the web application are permitted to execute on the database. *Applying diversity to the SQL language can accomplish this goal*. By applying N-variant diversity to the SQL language, the system can guarantee that an attacker cannot use a single payload to construct two differently diverse malicious SQL statements having the same structure.

7.2.3 Step 3 – Create the reexpression functions

Diversifying the SQL language is achieved by diversifying SQL keywords. This technique successfully diversifies the language because the ordering of keywords and literals defines the structure of an SQL statement. We could also have chosen to diversify literals, but these can be supplied by user input. The reexpression functions will therefore map SQL keywords to a new set of keywords. SQL comment syntaxes (`#, --`

`, or /*` in MySQL) are also reexpressed by the data sources because they can remove structure from an SQL statement.

The first reexpression function, R`, is chosen to be the identity function. $R``(\text{keyword}) = \text{keyword2}$, i.e. append the character ‘2’ to the keyword, was chosen as the second reexpression function because it is simple and ensures the disjointedness requirement.

The possibility of partial overwrites must also be considered when selecting the reexpression functions. Partial overwrites would be possible in this example by partially modifying keywords in constructed SQL queries. However, this method of attack would involve analyzing constructed SQL statements and modifying different locations within those statements for each variant. Because identical input is replicated to both variants, this is not considered a realistic attack scenario.

7.2.4 Step 4 – Determine inverse reexpression functions

$R`^{-1}$ is the identity function and $R``^{-1}(\text{keyword2}) = \text{keyword}$. This second inverse function strips away the ‘2’ that was appended by its associated reexpression function.

7.2.5 Step 5 – Identify data source and data sink

The protected information has already been identified as SQL statements. The only part of this system that processes SQL statements is the database. Therefore, the database is the data sink. The data source will be composed of all parts of the system that supply SQL statements to the database. For this system, only the web application (PHP script) supplies SQL statements to the database. The PHP script might contain vulnerabilities which would allow an attacker to manipulate the SQL statements issued to

the database. These vulnerabilities are already known to exist in at least one form as SQL injection attacks. These particular data sources and data sinks map cleanly from programs in the system to N-variant components. This is not necessary but it will make replicating the sources and sinks easier.

7.2.6 Step 6 – Determine data source reexpression mechanism and construct variant data sources

The nature of the PHP script allows a static transformation, and this is the preferred approach. All SQL keywords are hard-coded as strings in the script. There are no dynamically or externally obtained SQL keywords, and so these hard-coded strings can be statically transformed using the appropriate reexpression function.

Rex_S` and Rex_S`` are applied to the original PHP script to produce two variant scripts. Because R` is the identity function, $Rex_S$` is the identity transformation, and no changes are made to the original script for the first variant. $Rex_S$`` applies the R`` function to all SQL keyword strings encoded in the PHP script. This transformation was performed manually for this system, but it could be performed by an automated search and replace for larger scripts. The only operations on the SQL keyword strings were assignment and concatenation. These operations did not need to be transformed to preserve the semantics.

The PHP script in the original system was executed by an Apache web server. For the N-variant system, the web server was replicated so that each variant script was executed by a separate instance of Apache. This was not strictly necessary for the N-variant implementation, as the variant scripts could be executed simultaneously by a single server. But running the variant PHP scripts with separate instances of Apache

allows them to execute in separate memory spaces, thus protecting against Type III vulnerabilities between the data sources.

7.2.7 Step 7 – Determine data sink inverse reexpression mechanism and construct variant data sinks

Recall that the identified data sink was the database. The database could be replicated and serve as the data sink for each variant. However, there are practical issues with synchronizing two databases, especially if an attack would cause their contents to diverge. Therefore, an engineering decision was made to construct data sinks that will parse reexpressed SQL statements before they actually reach the database.

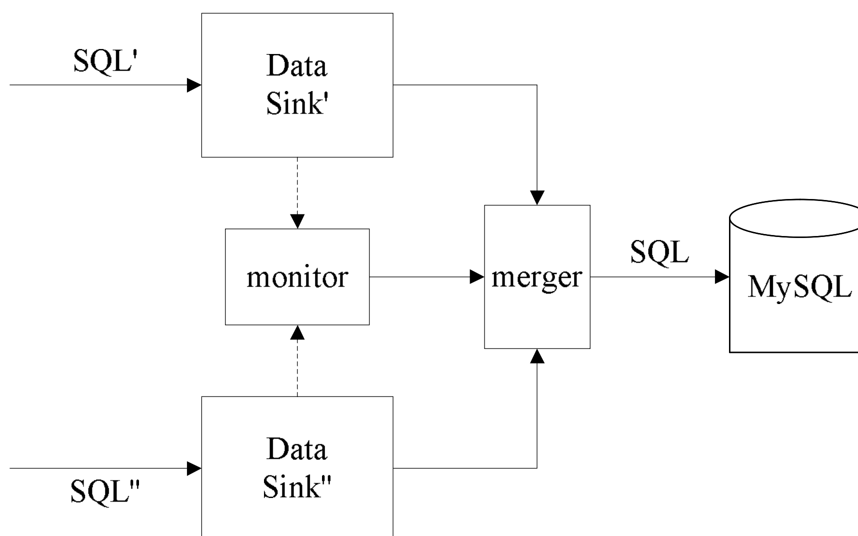

Figure 19. Data Sink Software Layer

A software layer was added between the web applications and the database. This layer contains the two SQL parser data sinks, a monitor, and an output merger to issue SQL statements to a single database. This software layer parses the variant SQL statements, checks for divergence, and under normal conditions (no attack), issues the original non-reexpressed SQL statement to the database. The purpose of this software

layer is to detect malicious SQL before it reaches the database and avoid having to synchronize multiple databases. An illustration of the data sink software layer is shown in Figure 19 for added clarity.

It is preferable to statically transform the data sink to operate with reexpressed data rather than to dynamically apply inverse reexpression at runtime. However, the data sink layer did not exist in the original system, and that means there is no reference data sink to directly transform into the variant data sinks. Thus, the first task is to create a data sink for the original system if this layer had existed. It is important to note that creating this original data sink is a consequence of the implementation decision to create the data sinks in front of a single database. This is not a general requirement for N-variant systems.

The data sink is an SQL parser that accepts an SQL statement as input and determines its structure. An SQL tokenizer is used to parse the statement and organize the result. The SQL tokenizer knows the grammar of SQL and parses a statement into tokens based on the layout of keywords and parameters in the statement. The obtained tokens are stored internally as an array. This array represents the structure of the SQL statement. Different token arrays represent different statement structures.

Java was used to create the data sink. The tokenizer is based on the SQL tokenizer from the SmallSQL Database Engine [6]. Tokens are hard-coded as strings in the program, and the tokenizer looks for these when it is scanning an SQL statement.

Once the original data sink is created, Inv_S` and Inv_S`` are applied to it in order to obtain the two variant data sinks. $Inv_S$` is the identity transformation and no changes were necessary to obtain the first data sink. $Inv_S$`` applies the R`` reexpresion function to

all search tokens in the tokenizer. This ensures that the second data sink will tokenize its SQL statement according to the reexpressed version of SQL. Both data sinks represent equivalent tokens the same way in the token arrays, and so under normal (non-malicious) inputs, they will produce identical token arrays for the same original SQL statement.

In order to coexist with the database on the same machine, the constructed data sinks listen for connections on different ports than the standard MySQL port (3306). In order to connect to the data sinks, the PHP scripts were slightly modified to send SQL statements on different ports.

7.2.8 Step 8 – Define semantics of divergence among the variants

The data sinks (SQL parsers) must receive logically equivalent SQL from their respective data sources. The equivalence of SQL statements can be validated by comparing the statement structures produced by the variant SQL parsers. A divergence is therefore defined as occurring when the data sinks receive SQL statements that have different structures.

7.2.9 Step 9 – Create the monitor

The monitor is created as a part of the data sink software layer. It monitors for a divergence by comparing the structures of SQL statements received by the data sinks. The monitor receives data from the sinks in the form of SQL token arrays.

The monitor waits until it receives data from both data sinks before performing an index-for-index comparison of the two SQL token arrays. If the token arrays are equivalent, then the monitor allows the SQL statement to proceed to the database. This is consistent with the monitor's property that it should not inhibit correct functionality

under normal (non-attack) situations. If the token arrays are not equivalent, then the monitor raises an alarm by returning a warning message back to the data sources in place of normal data results. The monitor prevents abnormal SQL from being issued to the database by signaling the output merger when a divergence has been detected.

The monitor initiates recovery by restarting the data sinks and terminating the execution of the PHP scripts. The data sinks do not store any state that persists between executions, so restarting them successfully puts them in an uncompromised state. The monitor terminates execution of the PHP scripts by returning the warning message. The scripts exit upon receiving this message.

7.2.10 Step 10 – Create the input splitter

The data sources receive input in the form of HTTP requests from external clients. Embedded in these requests might be user supplied parameters. The input splitter must replicate all HTTP requests identically to both data sources.

The input splitter is implemented as a simple server that listens for HTTP requests on the standard HTTP port (port 80). Any requests received by the input splitter are routed to both variant web servers. In order for the three servers to coexist on the same machine, the two variant servers listen for requests on ports 81 and 82. These ports are blocked from accepting external connections. Using these ports for the variant web servers requires the input splitter to insert a different port number into HTTP requests that it routes to them. This affects only the mechanism by which data is routed to the data sources, and does not alter the actual data received as inputs by the data sources.

7.2.11 Step 11 – Create the output merger

Two external entities receive output from the variants. The database receives SQL from the data sinks and clients receive query results from the data sources. Both of these entities expect merged outputs.

An output merger for SQL statements issued to the database is created in the data sink layer. This output merger is situated between the data sinks and the database. It exploits the fact that the first variant uses the identity reexpression function by issuing its SQL statement to the database and dropping the statement from the second data sink. Before issuing any output, it first checks that the monitor has successfully validated the equivalence of the two statements.

The output merger for client results is a part of the server that performs the input splitting. The input splitting server routes HTTP requests to the variant servers and receives results from both. It ensures that the two result sets are equivalent before returning one of those result sets to the actual client.

7.2.12 Step 12 – Verify functional correctness

The final step is verifying the functional correctness of the N-variant system. The system was tested for functional correctness by examining and comparing the results of various HTTP requests for both the original and N-variant versions of the system. Various sets of parameters were tested along with various queries constructed by the PHP scripts. For all cases, both systems returned identical results to the client and modified the contents of the database in the same ways.

Additionally, the data sinks were tested separately using various reexpressed SQL statement pairs. Tests of various structures and SQL keywords were performed. For all of these tests, the data sink tokenizers produced identical token arrays.

7.3 SQL N-Variant Model

An instantiation of the general N-variant model for this system is shown in Figure 20. This model is derived from the general model and shows the components that were constructed during the synthesis process.

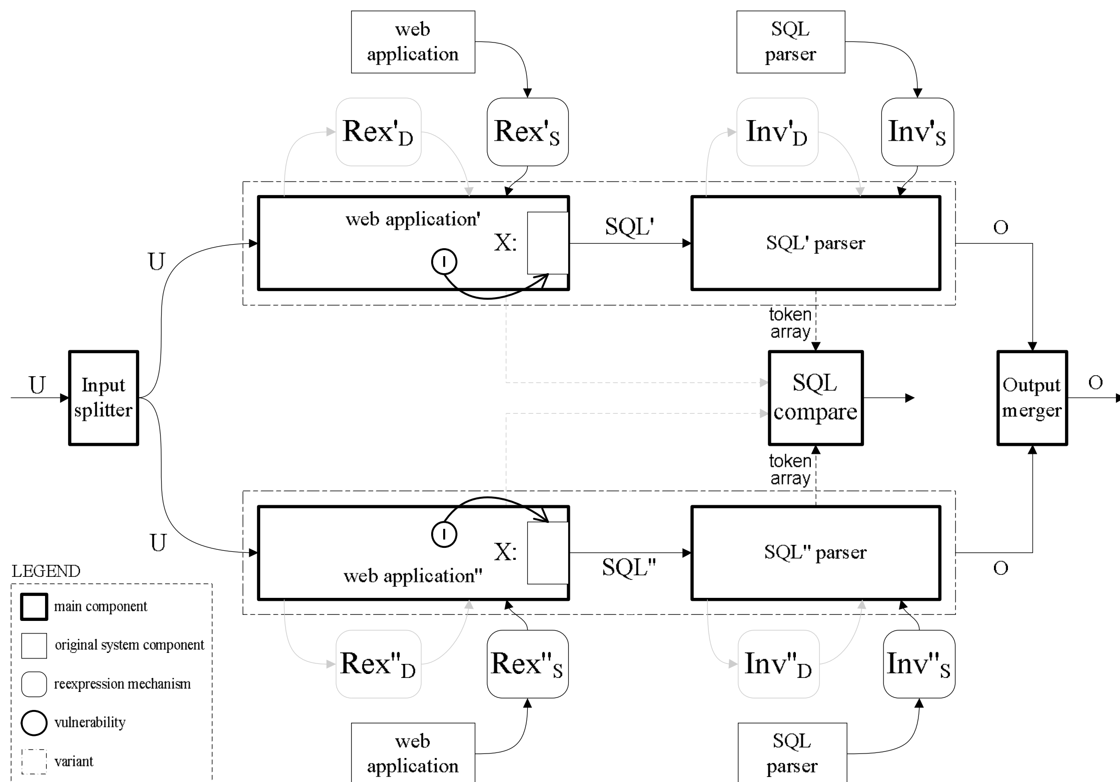

Figure 20. SQL N-Variant Model

7.4 Assessment

The SQL N-variant system must effectively ensure the desired security property that only intended operations are performed on the database. This was evaluated in terms of known vulnerabilities and an assessment of the monitor's divergence checking.

7.4.1 Known Vulnerabilities

The variant PHP scripts were constructed to retrieve credit card information from the database based on a user supplied name and password. The SQL query used to retrieve the information was:

```
SELECT Name,CardNum FROM credit WHERE
uid='$uid' AND pwd='$pwd';
```

The variables $uid and $pwd are supplied by the user and are intended to be inserted into the query as literals. However, SQL injection attacks can exploit this type of query construction by providing maliciously crafted inputs. We tested both the original system and the N-variant version of the system with three different sets of malicious input.

Our first test constructed malicious inputs designed to obtain the credit card information of a specific known user, "John", without providing a valid password. The supplied inputs were `"John' -- "` for $uid and `"anything"` for $pwd. The single quote escapes the first literal, effectively making 'John' the parameter that will be compared to uid values in the table. The `"--"` portion of the input comments out the remainder of the query. This removes the password comparison from the query and allows our attack to obtain the credit card information for "John" without supplying a valid password.

Our second test increased the attack seriousness. In this test, we constructed malicious inputs designed to obtain the credit card information of all users. The supplied inputs were `"' or 1=1; -- "` for $uid and `"anything"` for $pwd. The single quote escapes the first literal and makes the `"OR 1=1"` clause a part of the query, causing the WHERE clause to always evaluate to true. The remainder of the query is commented out by the `"--"`. These specially crafted inputs effectively alter the query to:

```
SELECT Name,CardNum FROM credit WHERE
uid='' OR 1=1;
```

This query will match each row in the table and return every name and credit card as results if it is allowed to execute on the database.

Our third test also constructed malicious inputs designed to obtain the credit card information of all users, but this attack did not rely on commenting out any part of the SQL query. The supplied inputs were `"' OR '' = '"` for both parameters. These inputs cause the resulting query to be:

```
SELECT Name,CardNum FROM credit WHERE
uid='' OR '' = '' AND pwd='' OR '' = '';
```

This query compares `''` (the empty string) to itself as part of both the uid and pwd comparison clauses. Because these evaluate to true, the uid and pwd comparisons do not need to succeed in order to match a row in the 'credit' table. Thus, the query will return every name and credit card in the table if it is allowed to execute on the database.

All three tests succeeded on the original version of the SQL system. The first test returned the credit card information for "John" and the second and third tests returned all names and associated credit cards. Executing these same tests on the N-variant version

of the system returned a "divergence detected" result in each case. Each attack was also attempted using reexpressed forms of the malicious input, e.g., using OR2. These attempts also failed to execute on the database and returned "divergence detected" results.

7.4.2 Divergence Monitoring Assessment

The above evaluation demonstrates how known SQL injection vulnerabilities are prevented by the N-variant system. However, there might be other vulnerabilities that could threaten the desired security property. The monitor's divergence checking was also assessed to show that malicious inputs to the data sinks are detected, regardless of how the inputs were generated, i.e., whatever Type I vulnerability was exploited in the data sources.

The data sinks and monitor were executed independently with malicious test inputs. The following types of malicious data were tested:

- **Incorrectly reexpressed statements:** We tested statement pairs that were both reexpressed using only one of the reexpression functions. Example tests: `"Select * from credit"`; `"Insert2 into2 credit values2 ('John Doe', '1234567898765432', 'JoDo', 'foo')"`.
- **Statements with entirely different structures:** We tested statement pairs that were correctly reexpressed but had different structures. Example test: `"Select CardNum from credit where uid='John' and pwd='foo'"` and `"Delete2 from2 credit where2 name='John'"`.
- **Statements with single mutations:** We tested statement pairs that were correctly reexpressed but had a single keyword difference. Example test: `"Select`

`CardNum from credit where uid='John' and pwd='foo'`" and "`Select2 CardNum from2 credit where2 uid='John' or2 pwd='foo'`".

- **Correctly reexpressed statements with different parameters:** We tested statement pairs that were equivalent except for different parameter text. Example test: "`Select CardNum from credit where uid='John' and pwd='foo'`" and "`Select2 CardNum from2 credit where2 uid='Bob' and2 pwd='bar'`". These statements are recognized as divergent because the parsers use the text of parameters as tokens.
- **Statements that are equivalent except for an extra clause appended to one:** We tested the monitor's ability to detect a divergence between two statements that are correctly reexpressed and equivalent except for an appended clause to one. Example test: "`Select * from credit where name='John'`" and "`Select2 * from2 credit where2 name='John' order2 by2 CardNum`".
- **Statements with incorrectly reexpressed SQL comments (`#, --, or /*`):** We tested statements that were correctly reexpressed except for included comment syntax. Example test: "`Select CardNum from credit where uid='John' -- and pwd='foo'`" and "`Select2 CardNum from2 credit where2 uid='John' -- and2 pwd='foo'`".

All tests on the above types of malicious data were detected as divergences by the monitor and were prevented from executing on the database.

7.5 Vulnerability Analysis

The SQL N-variant system addresses each of the four vulnerability types:

Type I: The construction of an N-variant system explicitly addresses Type I vulnerabilities. The tailored SQL diversity between the two variants ensures that an attacker cannot simultaneously and correctly manipulate the data flowing to both sinks. This guarantees that any attempt to modify existing SQL or construct malicious SQL queries will be detected.

Type II: The PHP scripts (data sources) were statically transformed to construct reexpressed SQL queries. The second vulnerability type is eliminated by this choice of static transformation over dynamic reexpression in the data sources.

Type III: The third vulnerability type is addressed by process isolation among the system components. The PHP script data sources are executed by two separate instances of the Apache web server running in distinct memory spaces. The data sinks also are executed within a separate process from the data sources. The defense against Type III vulnerabilities relies on the memory protection mechanisms afforded by the host operating system.

Type IV: The data sinks were created to parse SQL queries according to a set of reexpressed SQL keywords. These keyword sets were statically transformed for each variant data sink using the appropriate reexpression function. The fourth vulnerability type is eliminated by this choice of static transformation over dynamic inverse reexpression in the data sinks.

7.6 Comparison to Other Techniques

The N-variant system created in this chapter set out with a desired security goal of ensuring that only intended operations are performed on a database. Following the N-

variant creation process produced an N-variant system with general protection against attacks which violate the security property. This protection includes but is not limited to SQL injection attacks. There might exist other unforeseen attacks that will be prevented by this system because of the general protection it provides. Other techniques for protecting databases from the execution of malicious operations have been previously investigated. These approaches have specifically addressed SQL injection as the attack type and may or may not provide the same general protection as the N-variant system. These approaches are grouped into five categories and discussed below.

The recommended techniques for preventing SQL injection attacks are to filter user input or use parameterized statements to build queries. Correct application of these techniques can eliminate the possibility of SQL injection attacks. However, filtering user input can be overly restrictive in some cases and not strict enough in other cases. Additionally, it is easy to write unsafe code which interacts with a database while security concerns are overlooked. For these reasons, the above techniques are often not used or used incorrectly, and vulnerabilities continue to be discovered in real-world web applications [39].

Diversity based approaches modify SQL queries to make it difficult for an attacker to know the proper syntax to use when injecting SQL into the query. Boyd and Keromytis presented SQLrand [8]. SQLrand modifies the names of SQL keywords by appending a random key to the end of each keyword. This approach relies on the secrecy of the random key to be effective. If the key is leaked and an attacker learns it, then the attacker can simply modify his own SQL to successfully inject a malicious query. The

N-variant approach is secretless and does not rely on assumptions about an attacker not knowing a key.

Parsing validation techniques observe the effects of user input on the structure of an SQL query. These techniques exploit the fact that SQL injection attacks alter the intended structure of a database query. Buehrer et. al present a parse tree validation approach that parses SQL queries before and after user input and compares the trees prior to issuing a query to the database [10]. Much like SQLrand, however, their technique relies on a secret key. The secret key wraps user input to allow the validator to identify which segments of a query are user-provided. If an attacker gains knowledge about what key will be used, then he or she can escape the wrapper and masquerade user input as trusted SQL.

Nguyen-Tuong et. al [30] and Pietraszek & Vanden Berghe [31] use tainting to track untrusted data at runtime, and allow untrusted data only to be used to build literals in a query. These approaches are prone to false negatives if untrusted sources are not exhaustively identified. To address this issue, Halfond et. al use a positive tainting paradigm to track trusted data instead of untrusted data [22]. Their position is that false positives are preferable to false negatives. All three of these implemented techniques relied on either a modified platform or object-oriented language specific constructs. The N-variant approach does not require any language specific constructs or modifications.

Wasserman & Su provide a static analysis technique that uses FSA to model the syntactic structure of generated queries [44]. They automate the analysis segment of the process and, similar to parsing validation, model attacks as queries for which user input can alter the intended syntactic structure. Their specific implementation yielded a 20.8%

false positive rate and suggests that it is difficult to guarantee completeness statically. Code and technique complexities generally make it difficult for static techniques to prove completeness and correctness. Additionally, static analysis techniques have the disadvantage of significant programmer involvement. Each statically discovered vulnerability must be fixed by a programmer before deployment. N-variant systems provide runtime protection against all attacks in the given class, regardless of the specific vulnerability.

There are resource and performance disadvantages of the N-variant system compared to these other techniques. The redundant execution of two web servers combined with the addition of a data sink software layer and input splitter increases the amount of resources required. While these are essential elements of the N-variant framework, they might negatively impact performance. These concerns can be mitigated by using parallel hardware structures such as multiple processing cores.

Chapter 8: Conclusion

N-variant systems have several properties that make the N-variant architecture an advantageous security framework. The secretless property eliminates the practical difficulties of protecting secrets and removes the single point of failure that discovering a secret can create. The tailored diversity and redundant execution provide strong guarantees against entire classes of attack instead of specific vulnerabilities.

This thesis presented a general model for N-variant systems. This model characterized the components that make up an N-variant system and the security properties that each of these components has. Modeling N-variant systems has provided a general structure from which implementations can be derived and has permitted a detailed analysis of the framework. The model revealed vulnerabilities that exist within certain N-variant components. In particular, four types of vulnerabilities were identified to which all N-variant systems are subject and which all N-variant systems should address.

Instantiating the model for five existing N-variant systems demonstrated that the model can be used to analyze existing N-variant structures and to indicate their security properties. The four identified vulnerability types guided the security analysis. Component choices, particularly the reexpression functions and reexpression mechanisms, directly affect the system's susceptibility to each of these vulnerability types. Chosen reexpression functions should be carefully reviewed for their

susceptibility to partial overwrites and probabilistic attacks. The differences between static and dynamic application of these functions should also be carefully considered. This choice can affect the security guarantees afforded by an N-variant system.

In the future, the model and the construction process can be used as guides for building and evaluating N-variant systems. By following the synthesis steps and considering the properties elicited from the model, designers can better develop N-variant systems from the security properties they desire. The SQL example presented in Chapter 7 demonstrated the process for creating a new N-variant system from a desired security property. Building this system has shown that the N-variant construction process effectively guides the design and implementation of new systems. Using this process and knowledge of properties provided by the model, designers will be better able to construct future N-variant systems.

Bibliography

[1] P. E. Amman, J. C. Knight. Data Diversity: An Approach to Software Fault Tolerance. *IEEE Transactions on Computers*, v.37 n.4, p.418-425, April 1988.

[2] Apache Software Foundation. Apache HTTP Server project. http://httpd.apache.org.

[3] A. Avizienis and L. Chen. On the Implementation of N-Version Programming for Software Fault-Tolerance During Program Execution. *International Computer Software and Applications Conference*. 1977.

[4] E. Barrantes, D. Ackley, S. Forrest, T. Palmer, D. Stefanovic, and D. Zovi. Randomized Instruction Set Emulation to Disrupt Binary Code Injection Attacks. *ACM Computer and Communications Security*. 2003.

[5] E. Berger and B. Zorn. Diehard: Probabilistic Memory Safety for Unsafe Lanaguages. In *Programming Language Design and Implementation (PLDI)*, June 2006.

[6] V. Berlin. SmallSQL Database. http://www.smallsql.de, 2007.

[7] S. Bhatkar, D. DuVarney, and R. Sekar. Efficient Techniques for Comprehensive Protection from Memory Error Exploits. *USENIX Security*. 2005.

[8] S. W. Boyd and A. D. Keromytis. SQLRand: Preventing SQL Injection Attacks. In *Proceedings of the 2nd Applied Cryptography and Network Security (ACNS) Conference*, pages 292-302, June 2004.

[9] D. Bruschi, L. Cavallaro, and A. Lanzi. Diversified Process Replicae for Defeating Memory Error Exploits. *3rd International Workshop on Information Assurance.* 2007.

[10] G. T. Buehrer, B. W. Weide, and P. A. Sivilotti. Using Parse Tree Validation to Prevent SQL Injection Attacks. In *Proceedings of the International Workshop on Software Engineering and Middleware (SEM) at Joint FSE and ESEC*, September 2005.

[11] CERT. SQL Injection. http://www.us-cert.gov/reading_room/sql200901.pdf, 2009.

[12] L. Chen and A. Avizienis. N-Version Programming: A Fault Tolerance Approach to Reliability of Software Operation. *8th International Symposium on Fault-Tolerant Computing*. 1978.

[13] S. Chen, J. Xu, E. C. Sezer, P. Gauriar and R. K. Iyer. Non-control-data attacks are realistic threats. *14th USENIX Security*, 2005.

[14] M. Chew and D. Song. Mitigating Buffer Overflows by Operating System Randomization. *Tech Report CMU-CS-02-197*. December 2002.

[15] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. Pointguard: Protecting Pointers from Buffer Overflow Vulnerabilities. *USENIX Security*. 2003.

[16] B. Cox, D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, J. Knight, A. Nguyen-Tuong, J. Hiser. N-Variant Systems: A Secretless Framework for Security through Diversity. *15th USENIX Security*, August 2006.

[17] Jon Erickson. *Hacking: The Art of Exploitation*. No Starch Press. November 2003.

[18] H. Etoh. GCC Extension for Protecting Applications from Stack-Smashing Attacks. IBM, 2004. http://www.trl.ibm.com/projects/security/ssp.

[19] S. Forrest, A. Somayaji, and D. Ackley. Building Diverse Computer Systems. *6th Workshop on Hot Topics in Operating Systems*. 1997.

[20] M. Franz. Understanding and Countering Insider Threats in Software Development. *UC Irvine Technical Report ICS-TR-07-09*. 2007.

[21] D. Gao, M. Reiter, and D. Song. Behavioral Distance for Intrusion Detection. Recent Advances in Intrusion Detection. *8th International Symposium on Recent Advances in Intrusion Detection*. September 2005.

[22] W. Halfond, A. Orso, and P. Manolios. Using Positive Tainting and Syntax-Aware Evaluation to Counter SQL Injection Attacks. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, Portland, Oregon, November 2006.

[23] M. K. Joseph. Architectural Issues in Fault-Tolerant, Secure Computing Systems. *PhD Dissertation*. UCLA. 1988.

[24] J. Just, J. Reynolds, L. Clough, M. Danforth, K. Levitt, R. Maglich, and J. Rowe. Learning Unknown Attacks – A Start. *Recent Advances in Intrusion Detection*. October 2002.

[25] G. Kc, A. Keromytis, and V. Prevelakis. Countering Code-Injection Attacks with Instruction Set Randomization. ACM *Computer and Communications Security*. 2003.

[26] J. Knight and N. Leveson. An Experimental Evaluation of the Assumption of Independence in Multi-version Programming. *IEEE Transactions on Software Engineering*, Vol 12, No 1. January 1986.

[27] B. Littlewood and L. Strigini. Redundancy and Diversity in Security. *European Symposium on Research in Computer Security*. 2004.

[28] Lyu, M. (Ed). Software Fault Tolerance. *John Wiley & Sons Inc*. April 1995.

[29] A. Nguyen-Tuong, D. Evans, J. Knight, B. Cox, J. Davidson. Security through Redundant Data Diversity. *38th IEEE/IFPF International Conference on Dependable Systems and Networks (DSN)*, June 2008.

[30] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans. Automatically Hardening Web Applications Using Precise Tainting. In *Proceedings of the 20th IFIP International Information Security Conference*, 2005.

[31] T. Pietraszek and C. V. Berghe. Defending against Injection Attacks through Context-Sensitive String Evaluation. In *Proceedings of the 8th International Symposium on Recent Advances in Intrusion Detection (RAID)*, September 2005.

[32] J. Reynolds, J. Just, E. Lawson, L. Clough, R. Maglich, and K. Levitt. The Design and Implementation of an Intrusion Tolerant System. *Foundations of Intrusion Tolerant Systems (OASIS)*. 2003.

[33] B. Salamat, T. Jackson, A. Gal, and M. Franz. Orchestra: Intrusion Detection Using Parallel Execution and Monitoring of Program Variants in User-Space. *The European Conference on Computer Systems (EuroSys)*, 2009.

[34] F. Schneider and L. Zhou. Distributed Trust: Supporting Fault-Tolerance and Attack-Tolerance. *Technical Report TR 2004-1924, Cornell Computer Science Department*. January 2004.

[35] Kevin Scott and Jack W. Davidson. Safe Virtual Execution Using Software Dynamic Translation. *ACSAC*. December 2002.

[36] H. Shacham, M. Page, B. Pfaff, E. Goh, N. Modadugu, and D. Boneh. On the Effectiveness of Address-Space Randomization. *ACM Computer and Communication Security Symposium*. 2004.

[37] U. Shankar, K. Talwar, J. Foster, and D. Wagner. Detecting Format String Vulnerabilities with Type Qualifiers. *USENIX Security*, 2001.

[38] A. N. Sovarel, D. Evans, and N. Paul. Where's the FEEB?: The Effectiveness of Instruction Set Randomization. *USENIX Security*. 2005.

[39] Z. Su and G. Wassermann. The Essence of Command Injection Attacks in Web Applications. In *The 33rd Annual Symposium on Principles of Programming Languages (POPL 2006)*, Jan. 2006.

[40] Bjorn De Sutter and Koen De Bosschere. Introduction: Software techniques for Program Compaction. *Communications of the ACM, Vol 46, No 8*. August 2003.

[41] E. Totel, F. Majorczyk, and L. Mé. COTS Diversity Intrusion Detection and Application to Web Servers. *Recent Advances in Intrusion Detection*. September 2005.

[42] N. Tuck, B. Calder, and G. Varghese. Hardware and Binary Modification Support for Code Pointer Protection from Buffer Overflow. *International Symposium on Microarchitecture*. December 2004.

[43] D. Wagner and D. Dean. Intrusion Detection via Static Analysis. In *IEEE Symposium on Security and Privacy*, 2001.

[44] G. Wassermann and Z. Su. Sound and Precise Analysis of Web Applications for Injection Vulnerabilities. In *Conference on Programming Language Design and Implementation (PLDI)*, 2007.

[45] D. Williams, W. Hu, J. Davidson, J. Hiser, J. Knight, and A. Nguyen-Tuong. Security through Diversity: Leveraging Virtual Machine Technology. *IEEE Security and Privacy, Vol 7, No 1, pp 26-33*. 2009.

[46] J. Xu, Z. Kalbarczyk, and R. Iyer. Transparent Runtime Randomization for Security. *Symposium on Reliable and Distributed Systems*. October 2003.