DESIGN FOR SECURITY: MEASUREMENT, ANALYSIS AND MITIGATION TECHNIQUES

BY

SHUO CHEN

B.S., Peking University, 1997
M.E., Tsinghua University, 2000

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2005

Urbana, Illinois

ABSTRACT

Security vulnerabilities pose a serious threat to computer systems and network infrastructures. This dissertation addresses the measurement and analysis of security vulnerabilities and their impact, as well as the design of several techniques for vulnerability mitigation.

The research starts with the analysis of the security vulnerabilities published in the *Bugtraq* list and CERT advisories. An in-depth analysis of vulnerability reports and the corresponding source code of the applications motivates our development of a finite state machine (FSM) model to reason about security vulnerabilities. Most attacks can be decomposed into a series of violations of simple predicates, which provides a more formal way to depict these attacks. Besides the analysis of security vulnerabilities, many real-world security attacks are analyzed. The analysis shows that, although most current memory-corruption-based attacks compromise system security by overwriting control data, a new type of attack, namely non-control-data attack, can also be generally applicable to real-world software, e.g., they break into network servers to obtain the root privilege. This emphasizes the necessity of further research on defenses against memory-corruption-based attacks, including control-data attacks and non-control-data attacks.

The notion of *pointer taintedness* is introduced as the basis for detecting memory-corruption-based attacks. A pointer is said to be tainted if its value comes directly or indirectly from user input. Pointer taintedness allows the user to arbitrarily specify the target memory address to read, write, or transfer control to, which is usually a pathological program behavior. On the other hand, the attacker's ability to taint a pointer value is a crucial requirement for most attacks. Based on the notion of pointer taintedness, a theorem-proving technique is developed to identify potential security vulnerabilities via static source code analysis, and a processor architecture mechanism is implemented for dynamic pointer taintedness detection. Evaluations show that the proposed techniques offer better security coverage than existing methods: by detecting pointer taintedness, both control-data and non-control-data attacks are defeated in a unified manner.

TO EVERYONE IN MY FAMILY

ACKNOWLEDGMENTS

Five-years of study at Illinois is a long journey that is joyful and tough, but most importantly, fruitful. It is time to say thanks to the many people who gave me guidance, support, and encouragement.

First, I extend my deepest gratitude to my advisor, Prof. Ravi Iyer. In every stage of my thesis research, he gave me highly valuable advice about promising directions to pursue. His vision on the importance of investigating real security vulnerabilities was the source of my encouragement to accomplish the challenging data-analysis task, which turned out to be the most crucial component of this work. Prof. Iyer also spent a tremendous amount of time improving the quality of my papers. I am grateful that I have learned and grown in his research group.

I am fortunate to have Prof. Vikram Adve, Prof. Jose Meseguer and Prof. David Nicol on my doctoral committee. Their insightful comments about my thesis proposal inspired several new topics that I pursued in the past twelve months.

Dr. Zbigniew Kalbarczyk also supervised this thesis work. I deeply appreciate his great efforts on most of my papers – we often worked together on them word by word.

Many former and current students of the DEPEND group are my collaborators. Jun Xu, now an assistant professor in NCSU, offered me amazingly strong collaboration. I also thank Weining Gu, Karthik Pattabiraman, Nithin Nakka, and Keith Whisnant for wonderful collaborations.

Without a question, I should thank my whole family. They give me love and support every moment throughout the years.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Security vulnerabilities pose a serious threat to computer systems and network infrastructures. These vulnerabilities may result from design flaws, configuration errors, or implementation errors. Real-world vulnerabilities are due to a wide range of causes. Effectively defending against security attacks necessitates an in-depth analysis of historical data of known vulnerabilities so as to identify and extract their fundamental characteristics. A better understanding of security vulnerability characteristics can guide the development of effective defensive techniques applicable to a broad range of real-world applications.

This thesis is focused on measurement and analysis of security vulnerabilities; their impact and root causes, as well as the development of new design techniques for vulnerability mitigation. It specifically addresses three research questions: (1) What are the common characteristics of most security vulnerabilities? (2) What are the basic assumptions of current defensive techniques and what are the deficiencies of these assumptions? (3) How can we build better defensive techniques to overcome the deficiencies?

Analysis and Modeling of Security Vulnerabilities. To study the common characteristics of programming errors leading to security vulnerabilities, Bugtraq and CERT vulnerability databases are investigated along with the corresponding application source code. A finite state machine (FSM) model is developed to depict program states during a security attack as a series of primitive operations, each indicating a simple predicate that should be guaranteed by the application code. The FSM modeling methodology offers a representation approach with a higher degree of formalism in reasoning about security vulnerabilities. It is demonstrated that many categories of security vulnerabilities, including buffer overflow, format string bugs, heap corruptions, integer overflow, input validation errors, and file race conditions can be modeled as multi-stage FSMs consisting of logic predicates. The decomposability of most security

attacks suggests that attackers can potentially assemble new polymorphic attacks, and also that defenders can focus on checking the most common logic predicates to foil attacks. These two basic observations motivate the follow-on studies presented in this thesis.

Analysis of the Threat of New Attacks. Most real-world attacks exploit memory corruption errors in software to overwrite control data, including function pointers, jump pointers, and return addresses. These attacks are known as control-data attacks, which are widely understood and typically well documented [62]. Many defensive techniques, such as control data protection, non-executable memory pages, and system-call-based intrusion detection, have been proposed to defeat control-data attacks. The analysis of Bugtraq and CERT shows that many memory corruption vulnerabilities usually allow overwriting arbitrary memory locations. This indicates the possibility for non-control-data attacks as well, i.e., attacks that do not overwrite any control data but still cause security compromises, or *non-control-data attacks*. Although the theoretical possibility of non-control-data attacks has been suggested in previous work [24][73][77][83], it is generally believed that such attacks are rare against real-world software applications. A goal of this thesis is to show the general applicability of non-control-data attacks. Toward this end, such attacks are constructed against HTTP, FTP, SSH, and Telnet servers. These attacks corrupt configuration data, user identity data, user input strings, and decision-making flags, which evade the current detection mechanisms. Hence, non-control-data attacks represent a realistic security threat to operational software.

Static and Dynamic Defensive Techniques for Pointer Taintedness Detection. Defeating security attacks requires a definition of abnormal program behavior resulting from an attack. The notion of *pointer taintedness* is introduced as the basis to detect memory corruption attacks, including control-data attacks and non-control-data attacks. A pointer is said to be tainted if its value is derived directly or indirectly from user input. Pointer taintedness allows the user to arbitrarily specify the target memory address to read, write, or transfer control to, which is usually pathological program behavior. On the other hand, the attacker's ability to taint a pointer value is a crucial requirement for all types of memory corruption attacks. Based on the notion of pointer taintedness, a theorem-proving technique is proposed to identify potential security vulnerabilities via

static source code analysis. A processor-architecture-based solution for dynamic pointer taintedness detection is also proposed and implemented on a *SimpleScalar* processor simulator [13]. The implemented algorithm can effectively detect both control-data and non-control-data attacks, and therefore provide a more comprehensive detection capability than many intrusion detection methods.

In summary, the contributions of the research are two-fold: (1) demonstration of systematic approaches for analyzing and reasoning about system security and (2) design of effective and practical defensive techniques. This thesis research spans a broad range of security topics, including security vulnerabilities, security threat modeling/measurement, formal security properties verification, novel attacks, and ultimately defense techniques at processor architecture levels. It is unique because of the analysis-centric approach: a significant amount of effort is dedicated to analyzing real-world security vulnerabilities and uncovering deficiencies in current defensive techniques.

CHAPTER 2

RELATED WORK

The research work presented in this thesis combines three elements: the study of vulnerability data, the study of attacks, and the design of defensive techniques. This section provides a literature review for each of these topics individually and discusses how previous research projects motivate the workof this thesis. In addition, comparisons to the existing literature are made to highlight the uniquess of this work.

There has been substantial research in modeling, analysis, and classification of security problems, some of which is based on real data. Several studies have proposed classifications to abstract observed vulnerabilities into easy-to-understand classes. Representative examples include *Protection Analysis* [10], *RISOS* [1], Landwehr's taxonomy [42], Aslam's taxonomy [6], and the *Bugtraq* classification. Similarly, taxonomies for intrusions have been proposed. Examples include Lindqvist's intrusion classification [45] and the Microsoft *STRIDE* model [38]. In addition to providing taxonomies, [42] and [45] perform statistical analysis of actual vulnerability data, based on the proposed taxonomies. Several studies focus on modeling attacks and intrusions with the objective of evaluating various security metrics. Michael and Ghosh [48] employ an FSM model constructed using system call traces. By training the model using normal traces, the FSM is able to identify abnormal program behaviors and thus detect intrusions. In [67], an FSM-based technique to automatically construct attack graphs is described. The approach is applied in a networked environment consisting of several users, various services, and a number of hosts. A symbolic model checker is used to formally verify the system security. Recent studies have proposed stochastic models to quantitatively evaluate security metrics. Ortalo et al. [56] develop a Markov model to describe intruder behavior and evaluate system security in terms of METF (mean effort to failure). Madan [46] describes a semi-Markov model to evaluate an intrusion-tolerant system subject to security attacks. Several security and reliability metrics (e.g., METF and availability) are defined and shown to be solvable. Clearly, such a model requires that parameters, e.g.,

probabilities of transitions and sojourn time, be available or estimated. Despite these studies on security vulnerabilities, there is little work on modeling of discovered security vulnerabilities to capture how and why an implementation fails to achieve the desired level of security. The uniqueness of the analysis described in CHAPTER 3 is to use actual vulnerability data and code inspection to derive FSMs to describe program predicates, which provide a higher degree of formalism in reasoning about real-world security vulnerabilities.

The research on non-control-data attack described in CHAPTER 4 is motivated by a number of papers investigating system susceptibilities under hardware transient errors. It has been shown that random hardware faults can lead to security compromises in many real-world systems. Boneh et al. [12] show that the Chinese Remainder Theorem based implementation of the RSA crypto algorithm is vulnerable to any hardware/software errors during certain phases of the algorithm. The produced erroneous cipher text allows the attacker to derive the RSA private key. In [80], it is observed that even single-bit-flip transient errors in critical sections of server programs can cause false authentications. The experimental work in [19] shows that bit-flip errors in Linux kernel firewall facilities allow malicious packets to survive firewall packet filtering. In [36], Govindavajhala and Appel conduct a real physical fault injection experiment with a spotlight bulb heating the PC memory chips. The Java language type system can be subverted with high probability under this harsh condition. Although the results in the context of random errors may not demonstrate imminent security threats, they clearly indicate the high possibility of finding attacks other than control hijacking in real-world systems. Also related are papers discussing the possibility of evading system-call-based host IDSs by disguising traces of system calls. Mimicry attacks [77][78] cannot be detected by the IDS because the malicious code can issue system call sequences that are considered legitimate under the IDS model. The attacks proposed in [74] evade IDS detection by changing foreign system calls to equivalent system calls used by the original program. It should be noted that these mimicry attacks still corrupt program control and thus are defeated by many control flow integrity based techniques.

As for defensive techniques, both static compiler analysis and runtime detection techniques have been developed to defeat memory corruption attacks. Generic static

techniques such as SPLINT [28] and Extended Static Checking [21] can check whether the specified security properties are satisfied in program code. Another family of techniques is referred to as type-safety enforcement. Based on the observation that memory corruption attacks (including both control-data attacks and non-control-data attacks) need to violate the type-safety property, type-safe languages (e.g., Java) and several compiler techniques, such as CCured [52], Cyclone [40], and SAFECode [27], are designed to achieve type safety through reimplementing software or recompiling legacy programs. Despite recent progress, the following facts can affect the demonstratable success of these techniques: (1) rebuilding existing software with type-safety property requires a tremendous amount of effort and a lot of human knowledge; (2) type-safety of an application is usually achieved by hiding type-unsafe behaviors in low-level software components, such as Java virtual machine, C library, and OS kernel. Existing attacks have also succeeded in exploiting memory bugs in these components. Although researchers have started to work on the type-safety in low-level components, comprehensively deploying these techniques is a big paradigm shift in programming principles. How this shift can be compatible with the real software industry is not yet clear. Minimally speaking, making this shift is considered a heavy-weight shift that will take a long time.

Realizing such deployment difficulty, people also propose light-weight defensive techniques to provide security without reimplementing or recompiling software. Earlier techniques provided protection against specific types of attacks. Representative techniques include *StackGuard* [25] and *Libsafe* [7] to defeat stack buffer overflow attacks, and *FormatGuard* [23] to defeat format string attacks. Defensive techniques which randomize process memory layout to defeat security attacks, are proposed [9][57][82]. Although the principle is generic enough to foil most memory corruption attacks, there are still barriers in the implementation and deployment, e.g., the deployment on hardware architectures with a 32-bit address space has been found to be vulnerable to brute-force attacks [65].

The notion of taintedness was first proposed in the Perl programming language as a security feature. Inspired by this, static detection techniques SPLINT [28] and CQUAL [66] apply taintedness analysis to guarantee that user input data is never used as the

format string argument in printf-like functions. In both tools, taintedness is an attribute associated with C program symbols. A symbol becomes tainted only if an explicit C statement passes a tainted value to it by assignment, argument passing, or function return. For several types of real memory-corruption attacks, pointers are tainted without explicit C statements tainting program symbols. Unlike the pointer taintedness analysis approach presented here, these techniques do not have a memory model and thus cannot reason about the taintedness at the memory level.

Advances in computer architecture research have resulted in a number of techniques that are considered generic against all types of memory corruption attacks. *Secure Program Execution* [73] and *Minos* [26], which have been proposed most recently, are techniques to protect control data integrity. They rely on the definitions of spuriousness and integrity of data, which bear certain similarities to taintedness. However, these techniques are unable to defeat non-control-data attacks.

CHAPTER 3

MODELING SECURITY VULNERABILITIES

3.1. Overview of the Modeling Approach

Analysis of security vulnerabilities has typically been approached in one of two ways: (i) using real data to develop a classification and perform statistical analysis; examples include Landwehr's study on security vulnerabilities [42] and Lindqvist's study on intrusions [45], and (ii) providing a degree of formalism by modeling vulnerabilities and attack characteristics; representative work includes Ortalo's Markov model of UNIX vulnerabilities [56] and Sheyner's attack graph constructor [67]. Our study combines the two approaches: real data is analyzed, in conjunction with a focused source-code examination, to develop a finite state machine (FSM) model to depict and reason about security vulnerabilities.

This chapter presents a detailed analysis of vulnerability reports from typical data sources such as CERT [17] and *Securityfocus* [14]. The analysis shows:

- Exploits must pass through multiple *elementary activities*, at any one of which it can be foiled.
- Exploiting a vulnerability involves multiple vulnerable *operations* on multiple objects.
- Analysis of a given vulnerability along with examination of the associated source code allows us to specify the predicates that need to be met to ensure security.

These observations motivate the development of an FSM (Finite State Machine) modeling methodology capable of expressing the process of exploitation by decomposing it into multiple operations, each of which includes one or more elementary activities. Since each elementary activity is simple, it is feasible (using the data and the application code) to develop a predicate and a corresponding primitive FSM (pFSM) to represent the elementary activity. The pFSMs can then be combined to develop FSM models of vulnerable operations and possible exploits.

The proposed FSM methodology is exemplified by analyzing several types of vulnerabilities reported in the data: stack buffer overflow, integer overflow, heap overflow, file race condition, and format string vulnerabilities. These vulnerabilities include both those that can be exploited remotely (e.g., those impacting Internet servers) and those that can be exploited by local users (e.g., privilege escalation of a regular user to root). It should be noted that this family of vulnerabilities constitutes 22% of all vulnerabilities in the *Bugtraq* database. For the studied vulnerabilities, three types of pFSMs are identified that can be used to analyze operations involved in exploiting vulnerabilities and to identify the security checks to be performed at the elementary activity level.

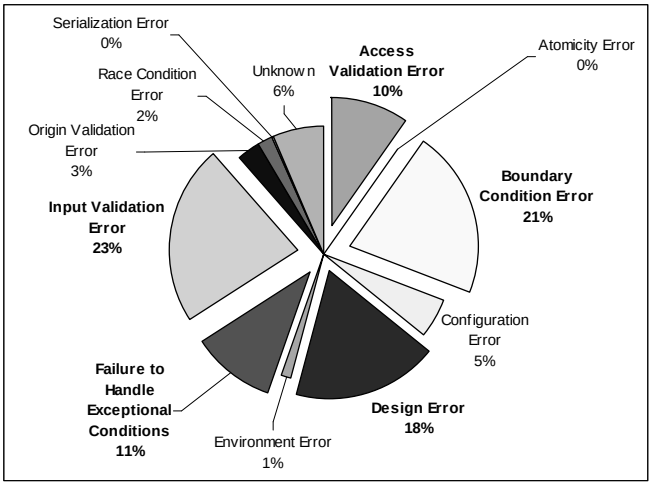
An additional demonstration of the usefulness of the approach was the discovery of a new heap overflow vulnerability now published in *Bugtraq* crediting the authors. The discovery was made when modeling another, known vulnerability

3.2. Analysis of the Bugtraq Database

3.2.1. Statistical Analysis

As of November 30, 2002, the *Bugtraq* database included 5925 reports on software-related vulnerabilities. Each vulnerability report in this database provides information such as version number of the vulnerable software, date of discovery, an assigned vulnerability ID, cause of the vulnerability, and possible exploits[1]. Figure 1 shows the breakdown of the 5925 vulnerabilities among the 12 defined classes. Observe that the pie-chart is dominated by five categories: input validation errors (23%), boundary condition errors (21%), design errors (18%), failure to handle exceptional conditions (11%), and access validation errors (10%). The primary reason for the domination of these categories is that they include the most prevalent vulnerabilities, such as buffer overflow (included under boundary-condition errors) and format string vulnerabilities (included under input-validation errors). The remaining categories, being very broadly defined (e.g., access validation errors, design errors), are more or less all-encompassing.

[1] Certain vulnerability reports in *Bugtraq* include exploits. For example, an exploit associated with vulnerability #5960 is provided in http://online.securityfocus.com/bid/5960/exploit



•*Access Validation Error*: an operation on an object outside its access domain.
•*Atomicity Error*: code terminated with data only partially modified as part of a defined operation.
•*Boundary Condition Error*: an overflow of a static-sized data structure: a classic buffer overflow condition.
•*Configuration Error*: a system utility installed with incorrect setup parameters.
•*Environment Error*: an interaction in a specific environment between functionally correct modules.
•*Failure to Handle Exceptional Conditions*: system failure to handle an exceptional condition generated by a functional module, device, or user input.
•*Input Validation Error*: failure to recognize syntactically incorrect input.
•*Race Condition Error*: an error during a timing window between two operations.
•*Serialization Error*: inadequate or improper serialization of operations.
•*Design Error* and, *Origin Validation Error*: Not defined.

Figure 1: Breakdown of Vulnerabilities and Definitions of Vulnerability Categories

3.2.2. An In-depth Analysis of Vulnerability Reports

An in-depth analysis of the data and information reported in *Bugtraq* together with a close examination of the associated application code is essential to understanding the root causes of the vulnerabilities. By examining the vulnerability reports and the associated application source codes, yielded three observations:

Observation 1: Exploits must pass through multiple elementary activities – at any one of which, one can foil the exploit. The scenario thus can be described as a serial chain in which each link (which is model as an elementary activity) provides a security checking opportunity: failure at any one elementary activity can foil the exploit.

This observation can be illustrated using data from three *signed integer overflow vulnerabilities* given in Table 1. Here the analysts have used three different activities as reference points to classify the same type of vulnerability into three categories, although there is nothing in the data to indicate the specific elementary activity corresponding to the observed vulnerability. Thus #3163 has been classified as input validation error, #5493 as a boundary condition error, and so on. The existence of three categories for the signed integer overflow vulnerabilities suggest that the code executions of the corresponding applications contain at least three activities: (1) get an input integer, (2) use the integer as the index to an array, and (3) execute a code referred to by a function pointer or a return address.

Table 1: Example of Ambiguity among Vulnerability Categories

Vulnerability	Description	Elementary Activity	Assigned Category
#3163 *Sendmail Debugging Function Signed Integer Overflow**	A negative input integer accepted as an array index	Get an input integer	Input validation error
#5493 *FreeBSD System Call Signed Integer Buffer Overflow*	A negative value supplied for the argument allowing exceeding the boundary of an array	Use the integer as the index to an array	Boundary condition error
#3958 *rsync Signed Array Index Remote Code Execution*	A remotely supplied signed value used as an array index, allowing the corruption of a function pointer or a return address	Execute code referred by a function pointer or a return address	Access validation error
* Each vulnerability reported to Bugtraq is assigned a unique ID, e.g., the report of vulnerability #3163 can be accessed from http://www.securityfocus.com/bid/3163.			

Data on *buffer overflow vulnerabilities* also indicates the existence of at least three potentially vulnerable activities: (1) get input string (#6157: interpreted as an *input validation error*), (2) copy the string to a buffer (#5960: interpreted as a *boundary condition error*), and (3) handle data (e.g., return address) following the buffer (#4479: interpreted as a *failure to handle exceptional conditions*). Again, each elementary activity provides an opportunity to apply a security check. For example, programmers can either check the input length in elementary activity 1, use boundary-checked string functions (e.g., *getns*, *strncpy*) in elementary activity 2, or deploy return address protection techniques, such as *StackGuard* [25] and *split-stack* [81], in elementary activity 3.

Similarly, an analysis of *format string vulnerabilities* (i.e., user's input strings containing format directives, such as `%n, %x, %d`) reinforces the validity of our observation: format string vulnerabilities are classified as *input validation error* (e.g., #1387 *wu-ftpd* remote format string stack overwrite vulnerability), *access validation error* (e.g., #2210 *splitvt* format string vulnerability), or *boundary condition error* (e.g., #2264 *icecast print_client()* format string vulnerability). Therefore, format string vulnerabilities also involve at least three elementary activities.

Observation 1 forms the basis of our FSM model. As will be seen in Section 3.3, each elementary activity can be modeled as a primitive finite state machine (pFSM) defined by a predicate which, if violated, results in an exploit. Multiple activities

performed on the same object form an operation, which is modeled as a FSM consisting of multiple pFSMs in series.

Observation 2: Exploiting a vulnerability involves multiple vulnerable operations on several objects. Let consider again the example *#3163 Sendmail debugging function signed integer overflow.* This vulnerability involves two operations: (a) manipulate the input integer (the object of this operation), consisting of elementary activity 1 (get an input integer) and elementary activity 2 (use the integer as the index to an array), and (b) manipulate the function pointer (the object of this operation), consisting of elementary activity 3 (execute a code referred by a function pointer).

Similarly, the vulnerability #5774 *Null HTTPD remote heap overflow vulnerability* involves three operations performed on three objects: (i) copying the oversized user input (the object) to a buffer allocated on a heap memory, which permits overwriting pointers following the buffer, (ii) freeing the buffer (the object), which allows writing a user-specified value to a user-specified location (e.g., function pointer), and (iii) executing the malicious code pointed to by the function pointer (the object). Aside from the heap overflow and signed integer overflow vulnerabilities shown here, stack buffer overflow and format string vulnerability also require multiple vulnerable operations. Thus following observation 1, since each operation can have multiple pFSMs, multiple operations will then be a chain of such pFSMs.

Observation 3: For each elementary activity, the vulnerability data and corresponding code inspections allow us to define a predicate, which if violated, results in a security vulnerability. For example, in the vulnerability #3163 *Sendmail debugging function signed integer overflow*, an integer index *x* is assumed to be in the range [0,100], but the implementation only checks to guarantee that $x \leq 100$, hence the problem (the vulnerability): allowing *x* to be a negative index and underflow an array. The correct predicate to eliminate this vulnerability would be $0 \leq x \leq 100$.

3.3. State Machine Approach to Vulnerability Analysis

The purpose of this section is to use the above observations to develop an FSM characterization of the vulnerable operations. The goal of this FSM is to reason whether the implemented operation, or more precisely each elementary activity within the

operation, satisfies the derived predicate. To this end, three steps are taken: (1) To represent each elementary activity as a primitive FSM (pFSM) expressing a predicate for accepting an input object. The predicate is first checked with respect to the specification and then with respect to the implementation. (2) To model an operation on an object as a series of pFSMs. (3) To cascade the operations to model the vulnerable implementation. While the objective here is to reason that a vulnerability (violation of a derived predicate) is not present in the implementation, we shall see that the process of this reasoning can allow uncovering a previously unknown vulnerability.

In order to show how a vulnerability can be analyzed using an FSM, consider the *Sendmail Debugging Function Signed Integer Overflow Vulnerability* (#3163). A signed integer overflow condition exists in writing the array `tTvect[100]` in the function `tTflag()` of *Sendmail* application. As a result, an attacker can overwrite the *global offset table (GOT)* entry[2] of the function `setuid()`[3] to be the starting point of attacker-specified malicious code (*Mcode*). Two operations are involved in exploiting this vulnerability: (1) writing debug level *i* to array location `tTvect[x]` (`i` and `x` are specified by the user) and (2) manipulating the *GOT* entry of function *setuid* (represented as `addr_setuid` for convenience in our description). The first operation consists of two pFSMs (activities): (i) $pFSM_1$ – get *i* and *x*, and (ii) $pFSM_2$ – write `i` to `tTvect[x]`. The second operation consists of a single $pFSM_3$ – call the function referred by `addr_setuid`. Recall that a pFSM represents a predicate for accepting an input object with respect to the specification and implementation. This is explicitly defined as follows:

Primitive FSM (pFSM). The primitive FSM consists of four transitions and three states. The transitions *SPEC_ACPT* and *SPEC_REJ* depict the specification predicates of accepting and rejecting objects (e.g., a user or a request), respectively. The transition *IMPL_REJ* represents the condition under which the implementation rejects what should be rejected according to the specification. This transition depicts the expected or correct behavior, i.e., the implementation conforms to the specification. A dotted transition *IMPL_ACPT* represents the condition under which an object that should be rejected according to the specification is accepted in an actual implementation. This transition is a

[2] The *GOT* entry is a function pointer to a specific function. Usually, in position-independent codes, e.g., shared libraries, all absolute symbols must be located in the *GOT* table, leaving the code position-independent. A *GOT* lookup is performed to decide the callee's entry when a library function is called.
[3] The published exploit chooses *setuid()* as the target function of *GOT* entry corruption, although the targets could be other functions.

hidden path representing a vulnerability. Three states are identified: (1) the *SPEC check state* (where an object is checked against the specification), (2) the *reject state* ⊗ – transition to reject state indicates that the object is insecure, according to the specification, and (3) the *accept state* ⍁ – transition to accept state indicates that the object is considered as secure object. See Figure 2.

Since each elementary activity is simple, it is feasible (using the data and the application code) to develop a predicate and a corresponding pFSM. The pFSMs can then be easily combined to depict FSM, modeling vulnerable operations and possible exploits.

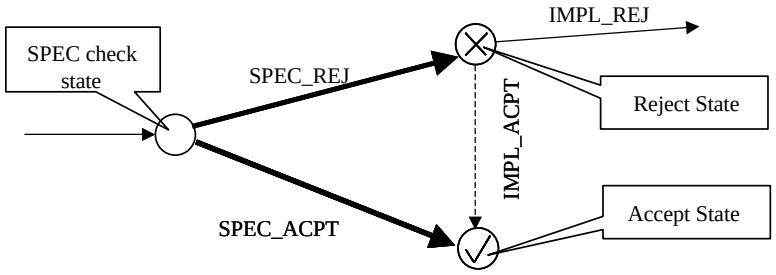


Figure 2: Primitive FSM (pFSM)

Figure 3 uses the semantic of the primitive FSMs and depicts the complete model of the process of exposing the *Sendmail Debugging Function Signed Integer Overflow Vulnerability*. As in a canonical FSM, a label *Condition*♦*Action* is associated with each transition. (Canonical FSM uses *Condition/Action* instead of the symbol ♦. Our modification is made because some of our examples need the slash symbol to represent filenames.) *Condition* refers to the condition for taking the transition, and *Action* is the action performed by the transition.

In the example (#3163), in Operation 1, elementary activity 1, the user inputs strings `str_x` and `str_i`, which are converted to signed integers *x* and *i*. The predicate of $pFSM_1$ specifies that if *str_x* represents an integer larger than 2^{31}, it should be rejected, i.e., $pFSM_1$ reaches the reject state, because signed integer `x` (4-byte variable) cannot correctly represent an integer larger than 2^{31}. (The signed integer `i` can also overflow, although it may not cause consequences as severe as an overflow of `x`). The real implementation does not check `str_x`, i.e., the transition of IMPL_REJ (marked by ?) does not exist, and the dotted transition (IMPLE_ACPT) is taken, allowing any `str_x` to arrive at the accept state of $pFSM_1$. At the object accept state, `str_x` and `str_i` are converted to signed integers `x` and `i`, which may become negative integers if overflow

occurs. The error exposed in $pFSM_1$ is that the system neglects checking the input `str_x`.

In Operation 1, $pFSM_2$ depicts the elementary activity *write* `i` *to* `tTvect[x]`. The predicate represented in $pFSM_2$ is the same as in the example in Observation 3, i.e., if an integer index `x` is in the range `[0,100]`, accept the `x`. However, the implementation checks only for the condition $x \leq 100$. As a result, negative `x` can be accepted and used in the operation `tTvect[x]=i` (arrive at termination state ◎). A potential security violation in Operation 1 is that the attacker can overwrite the GOT entry of `setuid()` so that it points to the location of a malicious code *Mcode.* Summarizing, Operation 1 consists of two pFSMs, each offering a security check, each, if provided, can foil an attack.

Operation 2 depicts the manipulation of the *GOT* entry corresponding to `setuid()` (i.e., `addr_setuid`). When *Sendmail* is started, `addr_setuid` is loaded to the memory. When `setuid()` is called, the value of `addr_setuid` is used as the function pointer to `setuid()`. Following the predicate depicted by $pFSM_3$, the system should check whether the value of `addr_setuid` is unchanged since it was loaded to the memory. If this is not the case (i.e., the `addr_setuid` has been tampered), the program should not call to the location indicated by the corrupted `addr_setuid`. However, the corresponding implementation of *Sendmail* does not perform the check on the `addr_setuid` (IMPL_ACPT=-♦- in $pFSM_3$), and accepts any value of `addr_setuid`. As a result, the program again makes the hidden (dotted) transition and the control jumps to the malicious code (*Mcode*) when `setuid()` is called.

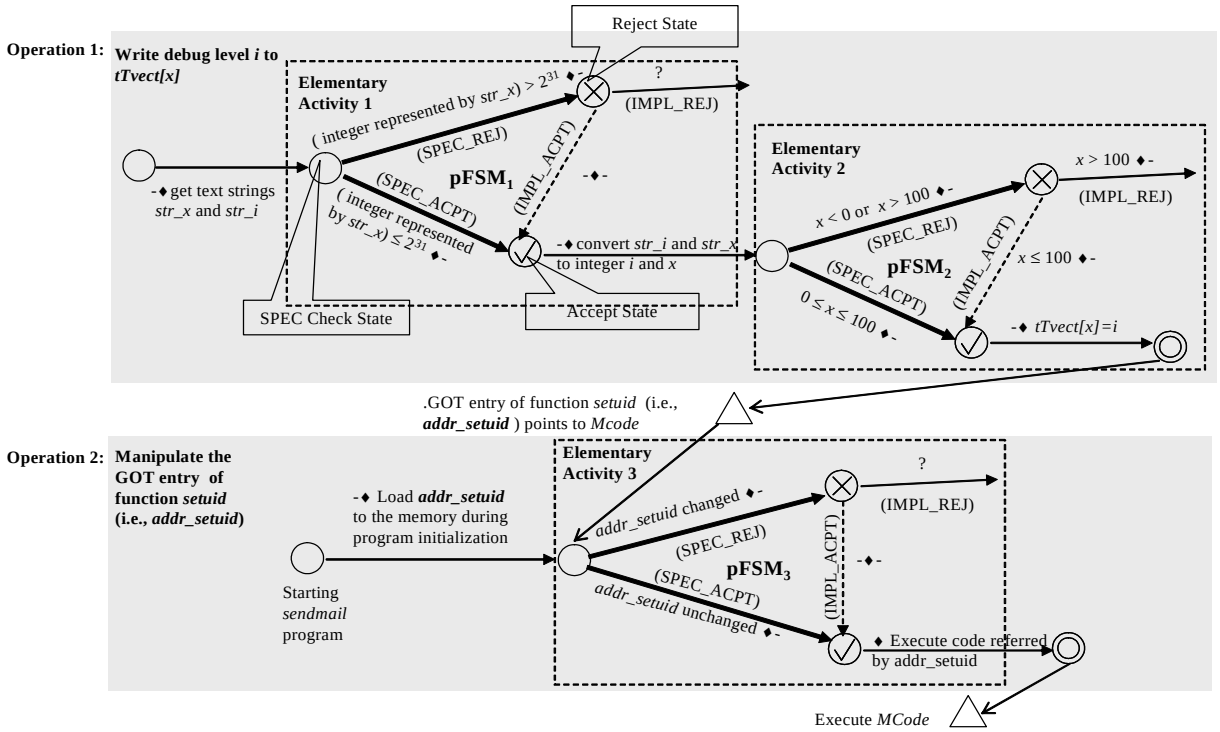


Figure 3: *Sendmail* Debugging Function Signed Integer Overflow Vulnerability

The FSM model introduces a notation of propagation gate (the triangle between FSMs) to depict the causality of the exploitation of the vulnerabilities in the two operations. For example, in Figure 3, exploiting *operation 1* (overwrite the `addr_setuid`) is the precondition of exploiting *operation 2* (execute *Mcode*), which is denoted by the upper propagation gate. The lower propagation gate (denoted as *Execute MCode*) can be the precondition for the exploitation in other operations.

3.4. Modeling Various Vulnerabilities Using an FSM

This section provides examples of applying the FSM approach to analyze security vulnerabilities. In each case, the predicates related to the elementary activities are determined by examining the vulnerability data and the corresponding source code of the applications in question.

3.4.1. Example 1: *NULL HTTPD* Heap Overflow Vulnerability

Null HTTPD is a multithreaded web server for Linux and Windows platforms. This software was chosen as an example because in the process of constructing the FSM model for the known vulnerability of *NULL HTTPD*, we discovered a new, as yet

unknown vulnerability (*Bugtraq* ID 6255). Discovery of the new heap overflow vulnerability demonstrates an additional potential of the FSM-based approach.

Null HTTPD 0.5 heap overflow is modeled as a series of four pFSMs shown in Figure 4a. $pFSM_1$ and $pFSM_2$ depict the buffer manipulation in the function `ReadPOSTData` (the function source code is shown in Figure 4b), which allocates a buffer (`PostData`, source code Line 1) and copies a user specified string from a socket (source code Line 4), which is marked as `input` in Figure 4a. One of the input parameters (`contentLen`) provides the length of `input`, which, by the specification[4], should be a non-negative integer. However, *Null HTTPD* allocates (by calling `calloc` in source code line 1) a buffer for `PostData` with size `1024+contentLen` without checking whether `contentLen` is non-negative. A buffer overflow occurs when the attacker provides a negative `contentLen` (e.g., `contentLen` = -800) to make `PostData` a buffer with only 224 bytes. This results in buffer overflow (denoted by $pFSM_1$) because *Null HTTPD* always copies at least 1024 bytes arriving from the socket to `PostData` (source code Line 4).

A New Vulnerability. Version 0.5.1 of *Null HTTPD* fixed the above overflow vulnerability by imposing the appropriate check to block a negative *contentLen* value before calling the function `ReadPOSTData` (this check is not shown in the source code of Figure 4b). Note that the socket programming style requires the users to specify the `contentLen` and `input` separately, because the socket has no way of determining the length of the input. The programmer must ensure that the length of `input` does not exceed the supplied `contentLen`.

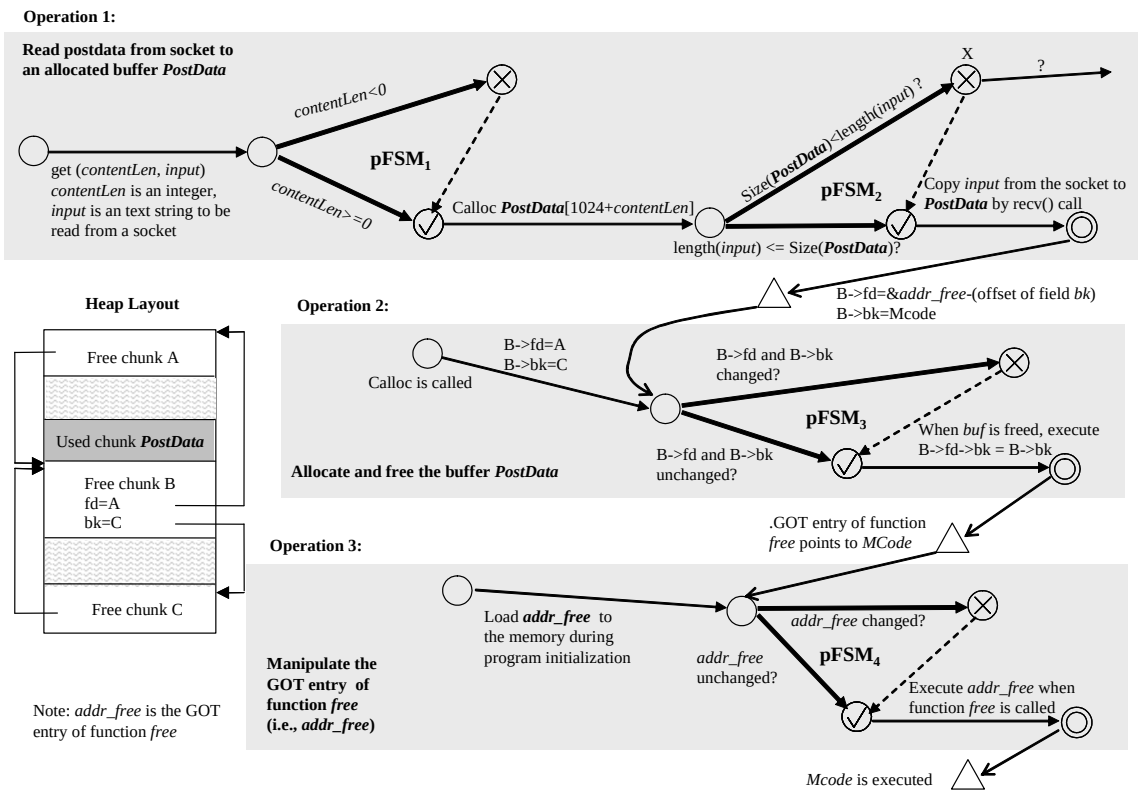
We now describe how constructing the FSM model for the known vulnerability leads to discovery of a new vulnerability for the same operation. $pFSM_1$ depicts the predicate to check `contentLen` against the specification. Similarly, $pFSM_2$ – the predicate to check the actual length of the supplied `input` – should reject `input` if its length is larger than allocated buffer size, i.e., it takes the transition marked "?". Source code Line 11 controls the termination condition of `recv` (source code Line 4). However, due to a logic error (`||` should be `&&` in source code Line 11), `recv` never terminates before the

[4] Although a well-defined specification does not exist, this particular specification can easily be deduced from the application.

entire *input* string is read from the socket. Thus, the outgoing transition (marked with a "?") from state *X* does not exist, and instead the hidden transition to the accept state ⓥ is taken. A malicious user can supply right `contentLen` but an arbitrary length string `input` to overflow the buffer `PostData`. Thus, constructing the FSM allowed us to uncover this new vulnerability.

As indicated earlier, each elementary activity offers an independent opportunity for checking. If the checks corresponding to the predicates depicted by $pFSM_1$ and $pFSM_2$ (in Figure 4a) are not in place, the impact of this vulnerability is further analyzed using $pFSM_3$, which describes the operation manipulating the heap layout (as shown in the left of Figure 4a). The buffer *PostData* is allocated on the heap, followed by a free memory chunk (chunk B). Free chunks are organized as a double-linked-list by GNU-libc. The beginning few bytes of each free chunk are used as the forward link (`fd`) and the backward link (`bk`) of the double-linked list. In this case, since free chunks A, B and C are in the list, `B->fd=A` and `B->bk=C`. The predicate defined in $pFSM_3$ provides a check so that `B->fd` and `B->bk` are not overwritten to an arbitrary value (i.e., $pFSM_3$ does not transit to the reject state), due to the overflow of the buffer `PostData` described in the $pFSM_1$ and $pFSM_2$. However, when the *PostData* is freed, the actual implementation does not check the pointer `B->fd` and `B->bk`, causing the transition from the reject state to the accept state (the hidden or dotted transition in $pFSM_3$), which allows the attacker to write an arbitrary value to an arbitrary memory location. Specifically, in this example, the attacker exploits this vulnerability and overwrites the GOT entry of the function `free()` so that it points to the location of malicious code *MCode*[5].

[5] Note that the assignment *B->fd->bk=B->bk* is executed when *PostData* is freed. We denote the *GOT* entry of *free()* as *addr_free*. The attacker sets *B->fd=&addr_free – (offset of the field bk)* and *B->bk=Mcode*, in order to make the *GOT* entry of *free()* pointing to *Mcode*.




```
1: PostData = calloc (contentLen +1024, sizeof(char)); x=0; rc=0;
2: pPostData= PostData;
3: do {
4:    rc=recv(sock, pPostData, 1024, 0);
5:    if (rc==-1) {
6:          closeconnect(sid,1);
7:          return;
8:    }
9:    pPostData+=rc;
10:   x+=rc;
11: } while  ((rc==1024) && (x<contentLen));
```

Figure 4: a) NULL HTTPD Heap Overflow Vulnerabilities b) Socket Reading Code

The $pFSM_4$ depicts the consequence of the corruption of the *GOT* entry of `free()` (i.e., `addr_free`), which is similar to the scenario depicted by $pFSM_3$ in the *Sendmail* vulnerability shown in Section 3.3. Finally, when the `free()` is called again, *Mcode* is executed.

In summary, this model consists of three operations. First operation encompasses two activities, each described by an independent pFSM ($pFSM_1$ and $pFSM_2$). Operation 2 and operation 3 consist of a single pFSM each. Cascading these four pFSMs allows us to reason through this entire vulnerable code.

The purpose of the next set of examples is two-fold: (1) show that FSM approach can analyze a broad class of vulnerabilities (specific examples relate to input validation

errors, file race condition errors, stack buffer overflow and format string vulnerability), and (2) provide additional examples of different types of pFSMs that broadly model the studied vulnerabilities.

3.4.2. Example 2: *xterm* Log File Race Condition

The program *xterm* emulates a terminal under the X11 window system. A file race-condition[6] exists when *xterm* writes messages to the user log file [8]. Figure 5 illustrates two pFSMs required to describe this vulnerability. Consider an example scenario: *xterm* needs to log Tom's messages to the log file `/usr/tom/x`. The predicate, which defines this operation, is depicted in $pFSM_1$, i.e., if Tom has no write permission or the provided filename is a symbolic link, the pFSM should reach the reject state ⊗. The real implementation follows $pFSM_1$, i.e., the reject condition of the predicate matches the implementation, hence this check is secure.

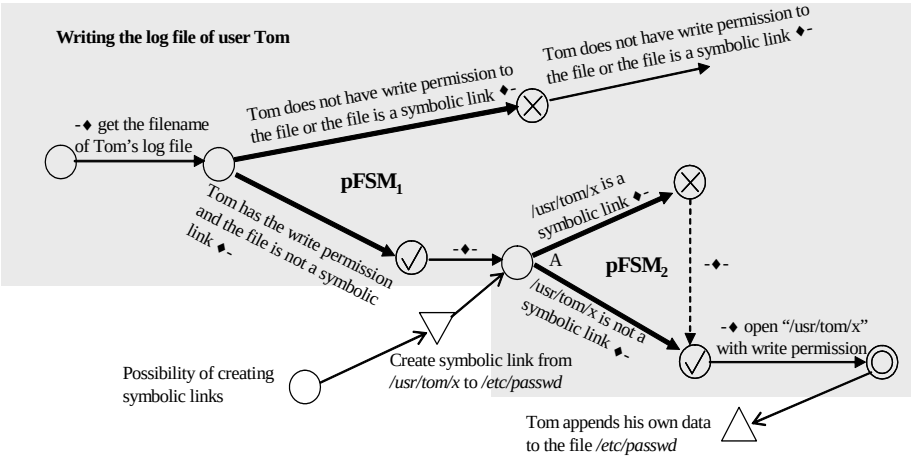


Figure 5: *xterm* Log File Race Condition

There is however a problem, which is analyzed in $pFSM_2$. In state A, Tom can delete the file `/usr/tom/x` and create a symbolic link from `/usr/tom/x` to `/etc/passwd`, so long as Tom creates the symbolic link before the system opens the file, i.e., a race condition exists. This timing problem is translated into a condition check in $PFSM_2$, which depicts the condition that Tom cannot create a symbolic link until the open operation is complete. As illustrated in this model, although there is no hidden path in $pFSM_1$, i.e., the implementation corresponding to $pFSM_1$ is secure, there is a hidden

[6] File race conditions are also referred as time-of-check-to-time-of-use vulnerabilities.

path in $pFSM_2$, indicating the possible race condition and the associated exploit: Tom appends his own data to the file `/etc/passwd`.

3.4.3. Example 3: Solaris *Rwall* Arbitrary File Corruption Vulnerability

Rwall is a UNIX network utility that allows a user to send a message to all users on a remote system (see [42] and CA-1994-06 in [17]). The file `/etc/utmp` on a remote system contains a list of all currently logged in users. *Rwall* daemon on the remote system uses the information in `/etc/utmp` to determine the users to which the message will be sent. A malicious user can edit the `/etc/utmp` file on the target system and add the entry "`../etc/passwd`". When the malicious user issues the command: `rwall hostname < newpasswordfile`, *Rwall* daemon writes the message (the `newpasswordfile`) to all terminals and to the file `/etc/passwd`.

In Figure 6, $pFSM_1$ checks if a given user has root privileges. The predicate dictates accepting the root user and rejecting a regular user (not having root privilege). In the real implementation, the write permission of the file `/etc/utmp` is set on, allowing a regular user to write this file (transition to the accept state). Specifically, as denoted by the propagation gate, a malicious user can add a "`../etc/passwd`" entry to the file `/etc/utmp`.

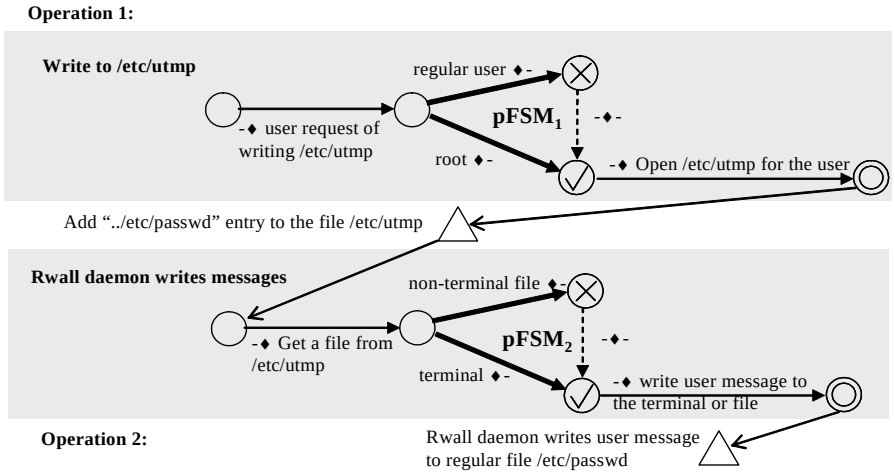


Figure 6: Solaris *Rwall* Arbitrary File Corruption Vulnerability

The Operation 2 depicts the message write operation performed by the *Rwall* daemon. The daemon gets a filename from the file `/etc/utmp`. The predicate represented by $pFSM_2$ states that if the filename refers to a non-terminal file, e.g.,

"`../etc/passwd`", it should be rejected, and if the filename refers to a terminal, e.g., "`/dev/pts/25`", the user-specified message should be written to the terminal.

In the implementation of the *Rwall* daemon, no file type check is performed. As a result, given an entry `/etc/passwd` added to the `/etc/utmp`, $pFSM_2$ transits to the reject state and ends up in the termination state ◎, which corresponds to a security violation – *rwall daemon writes user messages to regular file /etc/passwd*.

3.4.4. Example 4: Validation Error in *IIS* Decoding

CGI (Common Gateway Interface) programs under the directory `/wwwroot/scripts` are by design executable through HTTP request from a user. When *IIS*[7] receives a CGI filename request, it interprets the filepath as a path relative to `/wwwroot/scripts`. Therefore, unless the filepath contains "`../`", the target file should be under the directory `/wwwroot/scripts` (Bugtraq ID 2708).

In Figure 7, $pFSM_1$ depicts the predicate – if the target file does not reside in the directory `/wwwroot/scripts`, reject the request. Because the path is relative to `/wwwroot/scripts`, the above predicate is equivalent to – if the path of the target file does contain "`../`", reject the request. The *IIS* implementation includes two decoding steps. As illustrated in the $pFSM_1$, *IIS* implementation checks the following predicate – if the filepath contains "`../`"after the first decoding, reject the request. However, the implementation performs the second decoding step, which results in violating the predicate depicted by $pFSM_1$, and allows executing an arbitrary code (not residing in `/wwwroot/scripts`). This inconsistency between the predicate specified by $pFSM_1$ and the implemented predicate allows a transition from the reject state to accept state (the hidden path).

The attacker can thus supply a malformed filename containing sub-string such as "`..%252f`". After the second decoding, the string "`..%252f`" becomes "`../`"[8], which allows the execution of arbitrary programs, even those out of the directory `/wwwroot/scripts`. The worm Nimda and its variants actively exploit this vulnerability.

[7] *IIS* is Microsoft Internet Information Service.
[8] Note that "`%25`" is decoded to a character "`%`" and "`%2f`" is decoded to a character "`/`", so "`..%252f`" becomes "`..%2f`" after the first decoding, and is interpreted as "`../`" after the second decoding.

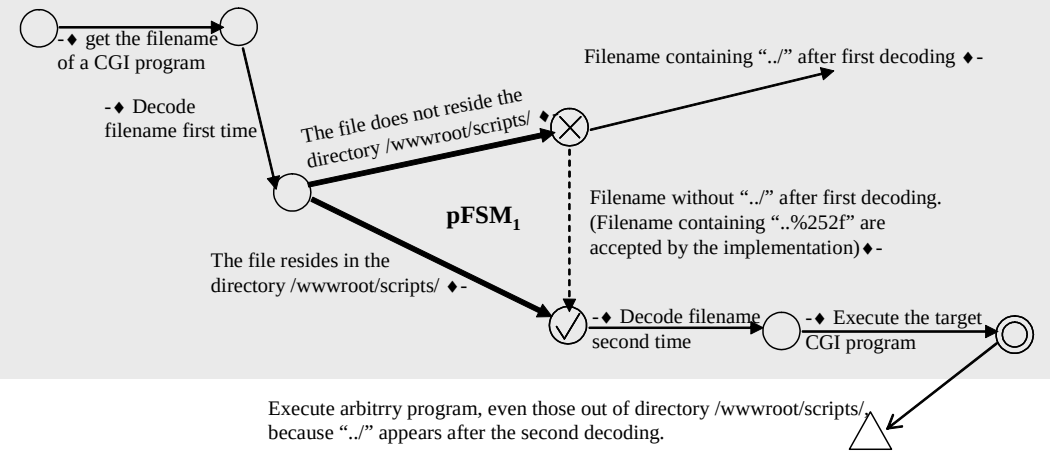


Figure 7: *IIS* Decodes Filenames Superfluously after Applying Security Checks

3.4.5. Stack Buffer Overflow Vulnerability and Format String Vulnerability

FSM is also used to model a stack buffer overflow vulnerability and a format string vulnerability.

Stack buffer overflow vulnerability (*#5960: GHTTPD Log() Function Buffer Overflow Vulnerability*): Two FSMs are constructed for this vulnerability. The first FSM describes a buffer overflow condition in the function `log()`: a buffer on the stack can be overflowed, which allows the return address of `log()`to be overwritten to the location of a malicious code Mcode. The second FSM describes the manipulation of the return address of `log()`: the return address is generated when `log()` is called, and it is consulted when `log()` returns. The corruption of return address in the first FSM may impact the second FSM, resulting in the execution of Mcode.

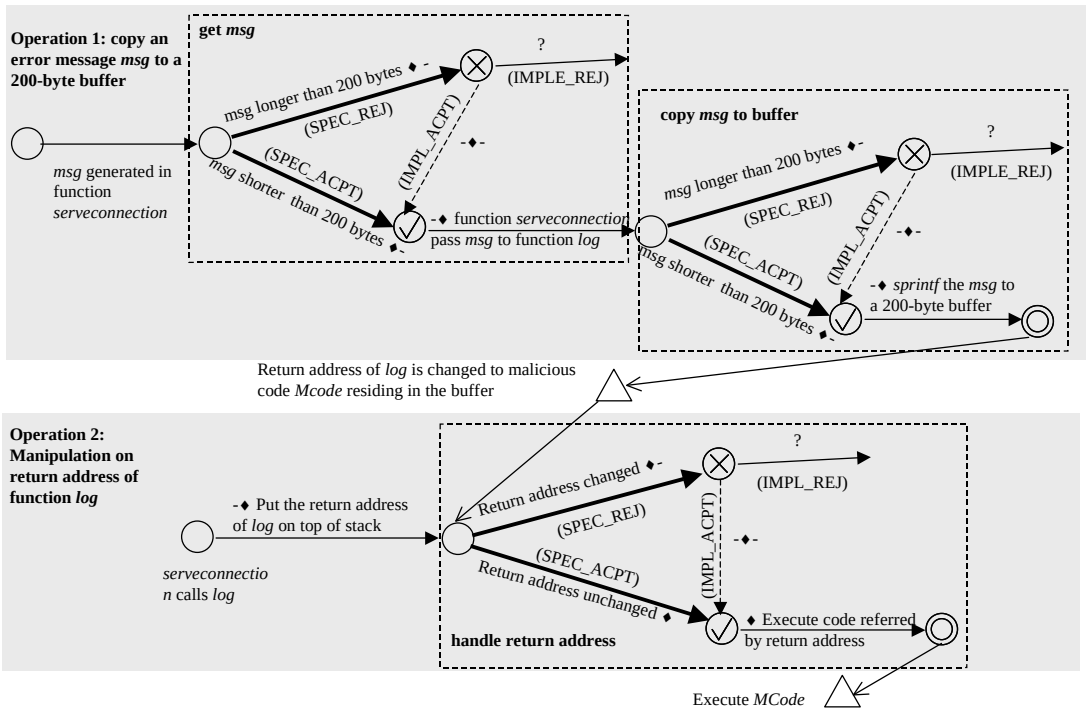


Figure 8: GHTTPD Log() Function Buffer Overflow Vulnerability

Format string vulnerability (*#1480 Multiple Linux Vendor rpc.statd Remote Format String Vulnerability*): Two FSMs are constructed. The first one describes *rpc.statd*'s failure in checking whether any format directives are embedded in the input string, which results in allowing the attacker to corrupt the return address of the function `vsyslog()` to Mcode. The second FSM describes the manipulation of the return address of `vsyslog()`, similar to the above example of #5960. This vulnerability also results in execution of Mcode.

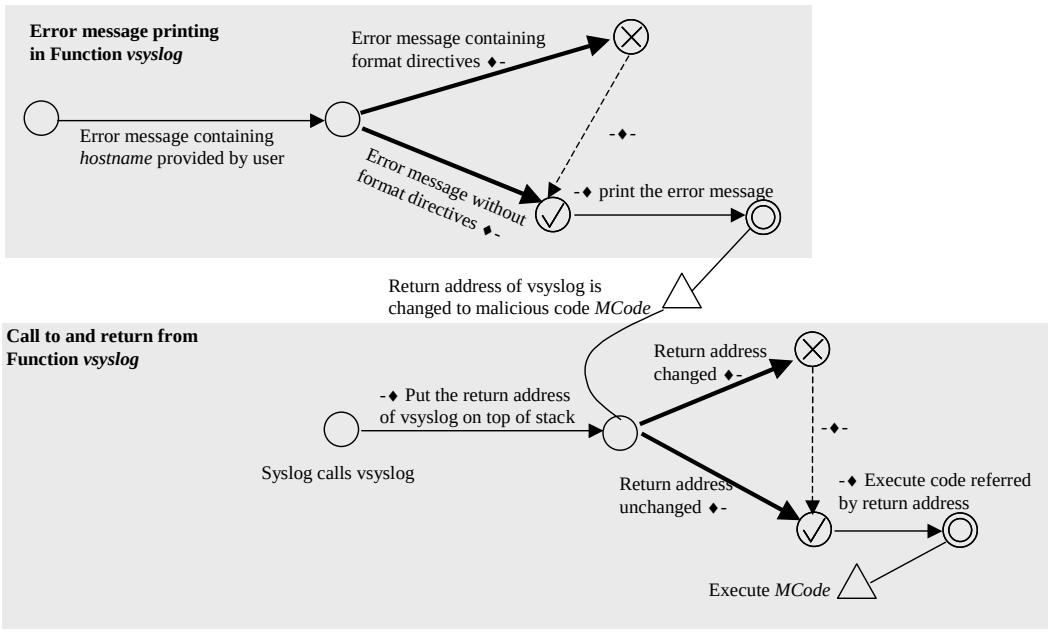


Figure 9: *rpc.statd* Remote Format String Vulnerability

3.5. Common Types of pFSMs

Examples in the previous sections show that the FSM approach enables a detailed modeling/analysis of several types of security vulnerabilities: buffer overflow, race condition, signed integer, and format string vulnerabilities (these four account for 22% of all vulnerabilities reported in *Bugtraq*). Vulnerabilities including, access validation errors, input validation errors, failure to handle exceptional conditions, can also be modeled, if the predicates are derived from available information vulnerability reports, exploits descriptions, and application source code.

As seen in the examples, the operations involving each vulnerability can be modeled as a series of pFSMs – each corresponding to an elementary activity. The simplicity of the predicates defining the pFSMs makes the generation of the overall FSM relatively easy. Since the pFSMs are critical to the analysis – it is meaningful to ask – Are there a few pFSMs, which allow us to model the bulk if not all of the studied data? The analysis shows that we only require three types of pFSMs to model the full range of studied vulnerabilities (i.e., stack buffer overflow, integer overflow, heap overflow, input validation vulnerabilities, and format string vulnerabilities).

Object Type Check. This is a predicate to verify whether the input object is of the type that the operation is defined on. In many circumstance, performing an operation on

an object of incorrect type results in *fail-secure* states [46], i.e., the operation fails without causing security to be compromised. For example, the object of a *ping* operation should be an *IP address* or a *hostname*. It is meaningless to say "`ping /etc/passwd`", because this will result in an error message "*unknown host /etc/passwd*". However, as we have seen in the examples, failure in *object type check* can be exploited by attackers, e.g., `rwalld` (see Figure 6) does not check whether the file type is a terminal or a non-terminal file, and *Sendmail* (see Figure 3) does not check whether the input represents an integer or a long integer.

Content and Attribute Check. This is a predicate to verify whether the content and the attributes of the object meet the security guarantee. Examples of *content and attribute checks* include (1) *IIS* filename decoding (Figure 7), where the program should verify that the request does not contain substring "`../`", (2) the system should check whether format directives are not embedded in the input, in order to prevent format string vulnerabilities (#1480), and (3) *GHTTPD* (#5960) should check whether the length of the input string is less than 200 bytes.

Reference Consistency Check. This is a predicate to verify whether the binding between an object and its reference is preserved from the time when the object is checked to the time when the operation is applied on the object. The examples include the return address referring to the parent function code, the function pointer referring to a function code, and a filename referring to a file. As shown in the FSM models, several conditions may result in violating the reference consistency, including stack smashing (#5960), signed integer overflow (Figure 3), heap overflow (Figure 4), format string (#1480), and file race condition (Figure 5).

The pFSMs representing the three generic predicates are depicted in Figure 10, which shows a typical operation (*P*) encompassing the three predicates. While all predicates may not be involved in all operations, the three suffice to model all the studied vulnerabilities classes (stack buffer overflow, integer overflow, heap overflow, input validation, and format string vulnerabilities).

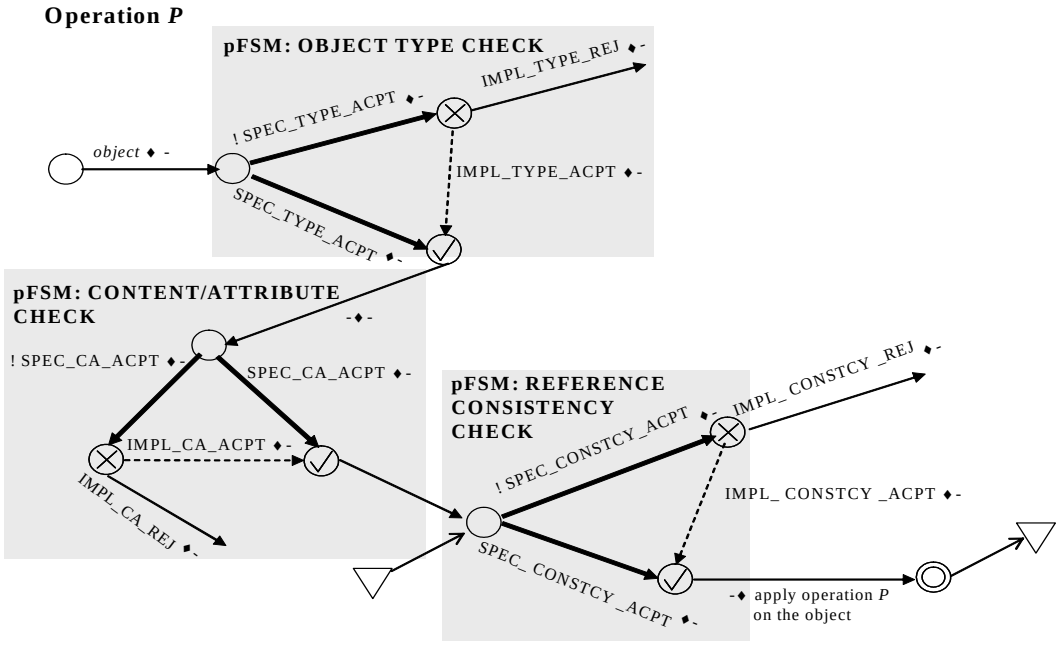


Figure 10: Types of Generic pFSMs

Table 2: Types of pFSMs

Type of pFSM / Vulnerabilities	Object Type Check	Content and Attribute Check	Reference Consistency Check
Sendmail Signed Integer Overflow	$pFSM_1$: Does the input represent a long integer?	$pFSM_2$: Is the integer in the interval [0 , 100] ?	$pFSM_3$: Is *GOT* entry of *setuid()* unchanged?
NULL HTTPD Heap Overflow		$pFSM_1$: contentLen ≥0? $pFSM_2$: length(input) ≤ size(buffer)	*$pFSM_3$: Are free-chunk links unchanged?* *$pFSM_4$: Is* GOT *entry of* free() *unchanged?*
Rwall File Corruption	$pFSM_2$: Is the target file a terminal?	$pFSM_1$: Does the user have a root privilege?	
IIS Filename Decoding Vulnerability		$pFSM_1$: Does the filename contain "../"?	
Xterm File Race Condtion		$pFSM_1$: Does the user have a write permission to the file?	$pFSM_2$: Does the filename refer to another unverified file?
GHTTPD Buffer overflow on Stack		$pFSM_1$: size(*message*) ≤ 200 ?	$pFSM_2$: Is the return address unchanged?
rpc.statd format string vulnerability		$pFSM_1$: Does the filename contain format directives (e.g., %n, %d)?	$pFSM_2$: Is the return address unchanged?

In Table 2, the pFSMs of the vulnerabilities analyzed in the previous sections are classified according to the three types of pFSMs identified above. The most common cause of the analyzed vulnerabilities is an incomplete content and/or attribute check. This

can be explained by fact that determining the correctness of an attribute (e.g., a buffer size) or a content (e.g., input contains a string "%n") of a given object may require a comprehensive understanding of the application. Incompleteness of a reference consistency check is another frequent reason for the vulnerabilities. While techniques protecting the return address have been widely recognized, very few techniques are available to protect other reference inconsistencies, such as inconsistency of function pointers, entries in GOT tables, and links to free memory chunks on the heap.

3.6. FSM Models Motivating Analysis and Mitigation of Attacks

Code-level FSM modeling is useful in understanding the internals of security vulnerabilities and attacks. The insights obtained from the analysis also help in identifying deficiencies of many current defensive techniques and formulating security predicates upon which to build security protection techniques. Based on the understanding of security vulnerability details, CHAPTER 4 presents a thorough analysis of real-world security attacks. The analysis shows that a type of attack, referred to as the non-control-data attacks, currently considered rare, is generally applicable against many real Internet servers. This finding exposes a serious deficiency in many defensive approaches. To address this threat, CHAPTER 5 and CHAPTER 6 show that a programming flaw, namely pointer taintedness, is a common cause of many vulnerabilities. A pointer is tainted when its value can be derived directly or indirectly from the user input. Since pointers are internal to applications, they should be transparent to users. Thus a tainted pointer is a potential security vulnerability. We will show a theorem-proving-based source code analysis and an architectural-level runtime detection technique to detect pointer taintedness.

CHAPTER 4

NON-CONTROL-DATA ATTACK: A REALISTIC THREAT

4.1. Control-Data Attacks versus Non-Control-Data Attacks

Most memory corruption attacks shown in CHAPTER 3, and in fact most known real attacks, follow a similar pattern known as the *control-data attack*: they alter the target program's control data (data that are loaded to processor program counter at some point of program execution, e.g., return addresses and function pointers) in order to execute injected malicious code or out-of-context library code (in particular, return-to-library attacks). The attacks usually make system calls (e.g., start a shell) with the privilege of the victim process. A cursory survey of the CERT/US-CERT security advisories [17][75] and the Microsoft Security Bulletin [49] shows that control-data attacks are considered to be the most critical security threats.

Because control data attacks are dominant today, many effective techniques have been proposed to defend against them. A reasonable question to ask is whether the current dominance of control-data attacks is due to an attacker's inability to launch non-control-data attacks[9] against real-world software. We think a possible answer is that attackers may in general be capable of mounting non-control-data attacks but simply lack the incentive to do so because control data attacks are usually easier to construct and require little application-specific knowledge. If this is indeed true, when the deployment of control flow integrity techniques makes control-data attacks impossible, attackers may have the incentives to bypass the defenses via non-control-data attacks.

It should be emphasized that the target question is not the *existence* of non-control-data attacks, but the *general applicability* of such attacks against real-world applications. Their existence has been suggested as a potential threat in previous work

[9] There are other names referring to attacks not altering control flow. For example, Pincus and Baker call them *pure data exploits* [60]. We use the term *non-control-data attack* mainly to contrast with control-data attack.

[24][73][77][83]. However, the applicability of these attacks has not been extensively studied.

This chapter provides experimental evidence to show that non-control-data attacks are realistic and can generally target real-world applications. The target applications are selected from the leading categories of vulnerable programs reported by CERT from 2000 to 2004 [17], including various server implementations for the HTTP, FTP, SSH, and Telnet protocols. The demonstrated attacks exploit buffer overflow, heap corruption, format string, and integer overflow vulnerabilities. All the non-control-data attacks constructed here resulted in security compromises as severe as those due to traditional control-data attacks – gaining the privilege of the victim process. Furthermore, the diversity of application data being attacked, including configuration data, user identity data, user input data, and decision-making data, shows that attack patterns can be diversified.

The experimental results show that attackers can indeed compromise many real-world applications without breaking their control flow integrity. The results also imply that finding a generic and secure solution to defeating memory corruption attacks is still an open problem when non-control-data attacks are considered. Many available defensive techniques are not designed for such attacks: some address specific types of memory vulnerabilities, such as StackGuard [25], Libsafe [7] and FormatGuard [23]; some have practical constraints in the secure deployments, such as pointer protection [24] and address-space randomization [9][57]; and others rely on control flow integrity for security, such as system call based intrusion detection techniques [29][30][31][34][32][37][63], control data protection techniques [26][73][54] and non-executable memory based protections [3][69]. Although compiler techniques such as CCured [52], Cyclone [40].and SAFECode [27] are promising and effective ways to enforce type safety, migrating existing large legacy code to the safe version is a non-trivial task. Therefore, it is important to address the realistic threat of non-control-data attacks in future research.

Besides the major contribution of demonstrating the general applicability of non-control-data attacks, the work in this chapter can also be viewed at a higher level as a step toward the empirical evaluation of defensive techniques. With more and more promising

defensive techniques being proposed, researchers have started to realize the necessity of such an empirical evaluation. In a survey paper [60], Pincus and Baker explicitly call for a thorough study on whether current defensive techniques "give sufficient protection in practice that exploitation of low-level defects will cease to be a significant elevation of privilege threat."

4.2. Applicability Claim of Non-Control-Data Attacks

While control-data attacks are well-studied and widely used, the current understanding of non-control-data attacks is limited. Although their existence has been suggested for a long time, e.g., Young and McHugh [83] gave an example of such attacks in a paper published even before the spread of the notorious Morris Worm[10] , the extent to which they are applicable to real world applications is not yet known. Because non-control-data attacks must rely on specific semantics of the target applications (e.g., data layout, code structure), their applicability is difficult to estimate without a thorough study of real vulnerabilities and the corresponding application source code. Control-data attacks, on the other hand, are easily applicable to most real-world applications once the memory vulnerabilities are discovered.

This work is partly motivated by results from a number of research papers investigating the impact of random hardware transient errors on system security. Boneh et al. [12] show that hardware faults can subvert an RSA implementation. Our earlier papers [19][80] indicate that even memory bit-flips in application can lead to serious security compromises in network servers and firewall functionalities. Govindavajhala and Appel conduct a physical fault injection experiment to subvert the Java language type system [36]. All these security compromises are very specific to application semantics, and not due to control hijacking. It should be noted, however, that the security compromises caused by hardware faults only suggest potential security threats since attackers usually do not have the power to inject physical hardware faults to the target systems. Nevertheless, the most compelling message from these papers is that real-world

[10] One of the attack vectors of the Morris Worm overruns a stack buffer in *fingerd* to corrupt a return address. This worm makes control-data attacks widely known to the public.

software applications are very likely to contain security-critical non-control data, given that *even random hardware errors can hit them with a non-negligible probability*.

We realize that several types of memory corruption vulnerabilities, in particular, format string vulnerability, heap overflow, signed integer overflow and double free vulnerabilities, are essentially memory fault injectors – they allow attackers to overwrite arbitrary memory locations within the address space of a vulnerable application. Compared to hardware transient errors, software vulnerabilities are more deterministic in that they always occur in the programs; and more amendable to attacks in that target memory locations can be precisely specified by the attacker. Based on this observation, we have the following claim:

> *Applicability Claim of Non-Control-Data Attacks*: *many real-world software applications are susceptible to non-control-data attacks, and the severity of the resulting security compromises is equivalent to that due to control-data attacks.*

The focus of our study is to empirically validate the applicability claim. Since this is a claim about real-world software, the only validation method we can imagine is to select a number of representative applications, and try to construct non-control-data attacks. Without real experiments, it would be impossible to answer three major questions: (1) what data within the target applications are critical to security other than control data, (2) do the vulnerabilities exist at appropriate stages of the application execution that can lead to eventual security compromises; (3) are the severity of the security compromises equivalent to those due to traditional control-data attacks.

4.3. Security Critical Non-Control Data

Corruption of application data can usually lead to crash, or incorrect computational results. While control data (e.g., return address, function pointers) are targeted by control-data attack, we are interested in finding non-control data that, if tampered, result in security compromises. In the study of several network server applications, we experimented with many types of non-control data. The results show that the following types of data are critical to application security:

- Configuration data

- User input
- User identity data
- Decision-making data

These classes are not meant to be mutually exclusive or collectively complete, but rather, the classification gives an organized reasoning about possibilities of non-control-data attacks. In this section, we explain each of these data types and why they are critical to security. For each data type, we describe the attack scheme(s) in Section 4.4 using real-world applications.

It should be noted that identifying security critical non-control data and construct corresponding attacks require more sophisticated knowledge about program semantics than that for control-data attacks. We currently rely on manual analysis of source code to obtain such knowledge.

Configuration Data. Site-specific configuration files are widely used by many applications. For example, many settings of the Apache web server can be configured using *httpd.conf* by the system administrator. The administrator can specify locations of data and executable files, access control policies for the files and directories, and other security and performance related parameters [5]. Similar files are used by FTP, SSH and other network server applications. Usually, the server application processes the configuration files to initialize internal data structures at the very beginning of program execution. At runtime, these data structures are used to control the behaviors of the application, and rarely change once the server enters the service loop. Corrupting configuration data structures allows the attacker to change and even control the behaviors of the target application. In our study, we have focused on the file path configuration information. The file path directives define where certain data and executable files are located so that the server can find them at runtime. They also serve as access control policies. In the case of a web server, the CGI-BIN path directive is not only used to locate the CGI programs, but also prevents a malicious client from invoking arbitrary programs, i.e., only a pre-selected list of trusted programs in the specific directory can be executed. If the configuration data can be overwritten through memory corruption vulnerabilities, an attacker can bypass the access control policy defined by the administrator.

User Identity Data. Server applications usually require remote user authentication before granting accesses. These privileged applications usually cache user-identity information such as user ID, group ID and access rights in memory while executing the authentication protocol. The cached information is subsequently used by the server for remote access decisions. If the cached information can be overwritten in the window from the time when the information is first stored in memory to the time when it is used for access control, the attacker can potentially change the identity and perform otherwise unauthorized operations within the target system.

User Input String. Changing user input is another way to launch a successful non-control-data attack. Input validation is a critical step in many applications to guarantee intended security policies. If user input can be altered after the validation step, an attacker would be able to break into a system. We use the following steps in the attack: (1) first, use a legitimate input to pass the input validation checking in the application; (2) then, alter the buffered input data to become malicious; (3) finally, force the application to use the altered data. The attack describe here is actually a type of TOCTTOU (Time of Check To Time of Use) attack: using legitimate data to pass the security checkpoint, and then forcing the application to use corrupted data that it considers legitimate. TOCTTOU was mainly described in the context of file race condition attacks in the existing literature. The attack studied here shows that the notion is applicable to memory data corruption as well.

Decision-Making Data. Network server applications usually use multiple steps for user authentication. Decision-making routines rely on several Boolean variables (conjunction, disjunction, or combination of both) to make the final verdict. No matter how many steps are involved in the authentication, eventually at a single point in the program control flow, there has to be a conditional branch instruction saying either yes or no to the remote client. Although such a critical conditional branch instruction may appear in different places in the binary code, each of them makes the critical decision based on a single register or memory data value. An attacker can corrupt the values of these final decision making data (usually just a one word Boolean variable) to influence the eventual critical decision.

Other Non-Control Data for Future Investigations. We have discussed four different types of data that if corrupted, can compromise security. Many other types of data are also critical to program security. We identify some of them for future investigations. File descriptors are integers to index the kernel table of opened files. They can point to regular disk files, standard input/output, and network sockets. If the attacker can change the file descriptors, the security of file system related operations can be compromised. Changing a file descriptor to 1 to that of a regular disk file could redirect terminal output to the file and result in severe security damages. Another possible target is the RPC (Remote Procedure Call) routine number. Each RPC service is registered with an integer as its index in the RPC callout link list. The caller invokes a service routine by providing its index. Malicious changes of RPC routine numbers could change the program semantics without running any external code.

4.4. Validating the Applicability Claim

This section validates the *applicability claim* stated earlier. It would be straightforward to manually construct vulnerable code snippets to demonstrate non-control-data attacks. This, however, does not validate the claim because what we need to show is the applicability of such attacks on *a variety of real-world* software applications. Toward this end, we need to first understand what applications are frequent targets of attacks and what types of vulnerabilities are exploited. A quick survey is performed on all 126 CERT security advisories between the year 2000 and 2004. There are 87 memory corruption vulnerabilities, including buffer overflow, format string vulnerabilities, multiple free and integer overflow. We found that 73 of them are in applications providing remote services. Among them, there are 13 HTTP server vulnerabilities (18%), 7 database service vulnerabilities (10%), 6 remote login service vulnerabilities (8%), 4 mail service vulnerabilities (5%) and 3 FTP service vulnerabilities (4%). They collectively account for nearly half of all the server vulnerabilities.

Our criteria in selecting vulnerable applications for experimentation are: (1) different types of vulnerabilities should be covered; (2) different types of server applications should be studied in order to show the general applicability of non-control-data attacks; and (3) there should be sufficient details about the vulnerabilities so that we can construct

attacks based on them. There are a number of practical constraints and difficulties. A significant number of vulnerability reports do not claim with certainty that the vulnerabilities are actually exploitable. Among the ones that do, many do not provide sufficient details for us to reproduce them. A number of the vulnerabilities that do meet our criteria are in proprietary applications. Experimenting with these close-source programs is time-consuming given the available manpower in our project. In addition, it is not possible to explain the vulnerabilities in source code. Therefore, we have used open-source server applications in our work for which both source code and detailed information about the vulnerabilities are available.

The rest of this section presents the experimental results. The demonstrated non-control-data attacks can be categorized along two dimensions: the type of security critical data presented in Section 4.3 and type of memory errors, such as buffer overflow and format string vulnerability. Although a significant portion of this section is to illustrate various individual non-control-data attacks in substantial details, the high level goal is to show that many real-world applications are in general vulnerable to such attacks.

4.4.1. Format String Attack against User Identity Data

WU-FTPD is one of the most widely used FTP servers. The *Site Exec Command Format String Vulnerability* [16] is one that can result in malicious code execution with root privilege. All the attack programs we obtained from the Internet overwrite return addresses or function pointers to execute a remote root shell.

Our goal is to construct an attack against user identity data that can lead to root privilege compromise without injecting any external code. Our first attempt was to find data items that if corrupted, could allow the attacker to login to the system as root user without providing correct password. We did not succeed in this because the SITE EXEC format string vulnerability occurs in a procedure that can only be invoked after a successful user login. That means an attacker could not change data that would directly compromise the existing authentication steps in FTPD. Our next attempt was to explore the possibility of overwriting the information source that is used for authentication. In Unix-based systems, user names and user IDs are saved in a file named `/etc/passwd`, which is only writable to a privileged root user. A natural thought is to corrupt

information in this file in order to get into the system. By overwriting an entry in this file, an attacker can later legitimately login to the victim machine as a privileged user. We observed that after a successful user login, the effective UID (EUID) of the FTPD process has properly dropped to the user's UID, so the process runs as an unprivileged user. Therefore, `/etc/passwd` can be overwritten only if we can escalate the privilege of the server process to root privilege. This is possible because the real UID of the process is still 0 (root UID) even after its EUID is set to be the user's UID. The success of the attack depends on whether we can corrupt certain data structure so that the EUID can be reverted to 0. FTPD uses the `seteuid()` system call to change its EUID when necessary. There are 18 `seteuid(0)` invocations in the WU-FTPD source code, one of which appears in function `getdatasock()` shown in Table 3. The function is invoked when a user issues data transfer commands, such as `get` (download file) and *put* (upload file). It temporarily escalates its privilege to root using `seteuid(0)` in order to perform the `setsockopt()` operation. It then calls `seteuid(pw->pw_uid)` to drop its privilege. The data structure `pw->pw_uid` is a cached copy of the user ID saved on the heap. Our attack exploits the format string vulnerability to change `pw->pw_uid` to 0, effectively disabling the server's ability to privilege dropping after it is escalated. Once this is done, the remote attacker can download and upload arbitrary files from/to the server as a privileged user. The attack compromises the root privilege of FTPD without diverting its control flow to execute malicious code.

Table 3: Source Code of `getdatasock()`

```
FILE * getdatasock( … ) {
  ...
  seteuid(0);
  setsockopt( ... );
  ...
  seteuid(pw->pw_uid);
  ...
}
```

The attack has been successfully tested on WU-FTPD-2.6.0. We first establish a connection to the control port of FTPD, and correctly login as a regular user Alice. FTPD sets its effective user ID to that of Alice (e.g., 109). The client then sends a specially constructed `SITE EXEC` command to exploit the format string vulnerability which overwrites the `pw->pw_uid` memory word to 0. The client then establishes the data

connection, and issues a *get* command which invokes function `getdatasock()`. Due to the corruption of `pw->pw_uid`, the execution of the function sets the EUID of the process to 0 permanently. The client can therefore download `/etc/passwd` from the server, add any entry desired, and then upload the file to the attacked server. An entry such as "`alice:x:0:0::/home/root:/bin/bash`" indicates that Alice can login to the server as a root user anytime via FTP, SSH or other available service. Figure 11 gives the state transition and flowchart of the attack.

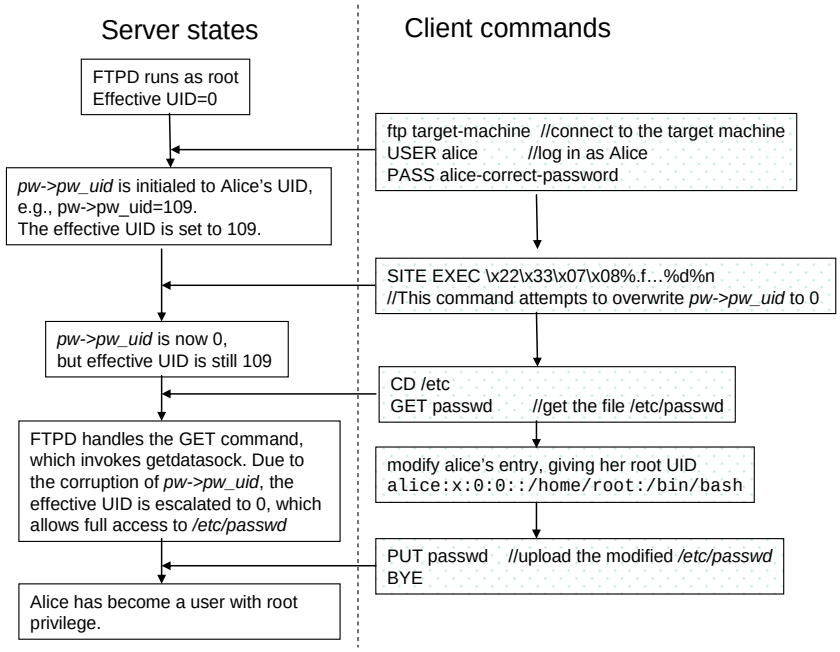


Figure 11: User Identity Data Attack in WU-FTPD

4.4.2. Heap Corruption Attacks against Configuration Data

We have found that the memory corruption vulnerabilities on an HTTP daemon and a Telnet daemon allow configuration data attacks to succeed in getting root shells, if these daemons run as root. Note that some HTTP daemons can run as an unprivileged user, e.g., a special user *nobody*, in which case the root compromise is unlikely to accomplish, no matter whether the attack is a control-data attack or a non-control-data attack. This fact does not affect our applicability claim stated before, because all we claim is that non-control-data attacks can get the same privilege level as control-data attacks, which is the privilege level of the victim server.

Attacking Null HTTPD. *Null HTTPD* is a multi-threaded web server on Linux. Two heap buffer overflow vulnerabilities have been reported [55]. Available exploit programs overwrite a Global Offset Table[11] (GOT) entry of a function when the corrupted heap buffer is freed. The program control jumps to the attacker's malicious code when a subsequent invocation of the function is made.

We found that corrupting CGI-BIN configuration string can result in root compromise without executing any external code. CGI (Common Gateway Interface) is a standard for running executables on the server for data processing. As explain in Section 4.3, CGI-BIN directive restricts a user from executing programs outside the CGI-BIN directory and is thus critical to the security of the HTTP server. A client's URL requesting the execution of a CGI program is always relative to the CGI-BIN configuration. Assume that CGI-BIN path of the server `www.foo.com` is `/usr/local/httpd/cgi-bin`, when a request of URL `http://www.foo.com/cgi-bin/bar` is processed, the HTTP server prefixes the CGI-BIN to *bar*, and executes the file */usr/local/httpd/cgi-bin/bar* on the server's file system.

Our attack is to corrupt the CGI-BIN configuration so that the shell program */bin/sh* can be started as a CGI program. The heap buffer overflow vulnerability is triggered when a special POST command is received by the server. Due to the nature of heap corruption vulnerability, an attacker usually can only precisely control the first two bytes[12] in the corrupted word at a time to avoid segmentation fault. We issue two POST commands to precisely overwrite four characters in the CGI-BIN configuration so that it is changed from "*/usr/local/httpd/cgi-bin\0*" to "*/bin\0*". After the corruption, we can start `/bin/sh` as a CGI program and send any shell command as the standard input to `/bin/sh`. For example, by issuing `rm /tmp/root-private-file` command, we observe that the file `/tmp/root-private-file`, writable only to the root, was removed. This indicates that we are indeed able to run any shell command as root, i.e., the attack causes the root compromise. Figure 12 shows the attack process.

[11] Global Offset Table (GOT) is a table of function pointers for calling dynamically linked library functions.
[12] If the value to be written is a valid address, four bytes can be overwritten by a single heap corruption attack.

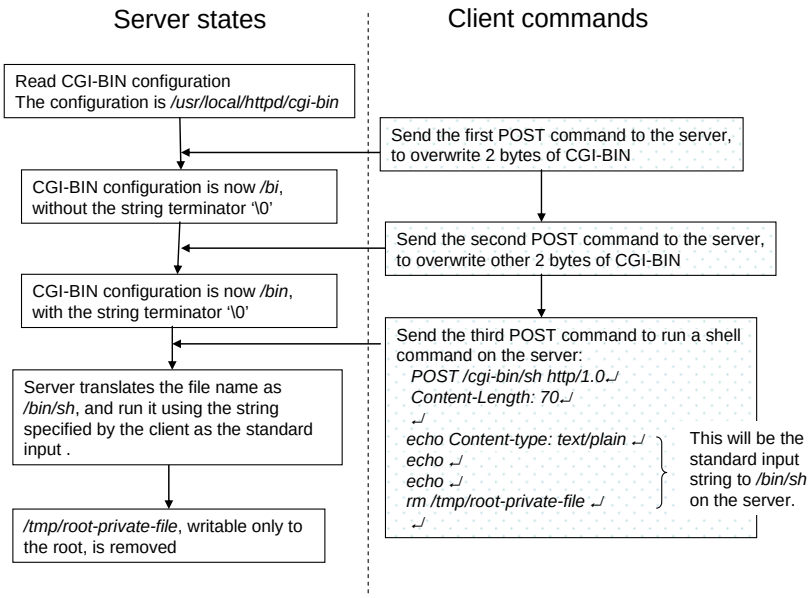


Figure 12: Configuration Data Attack against NULL HTTPD

Attacking NetKit Telnetd. A heap overflow vulnerability exists in many Telnet daemons derived from the BSD Telnet daemon, including a default RedHat Linux daemon *NetKit Telnetd* [15][51]. The vulnerability is triggered when function `telrcv()` processes client requests of '*AYT*' (i.e., *Are-You-There*) configuration. The attack downloaded from *Bugtraq* overwrites a GOT entry to run a typical malicious code starting a root shell.

When the daemon accepts a connection from a Telnet client, it starts a child process to perform user authentication. The file name of the executable for the authentication is specified by a configuration string `loginprg`, whose value can be specified as a command line argument. A typical value is `/bin/login`. Suppose the remote user is from `attacker.com`, function `start_login(host)` shown in Table 4 starts the command `/bin/login –h attacker.com –p` by making an `execv` call to authenticate the user. The integrity of `loginprg` is critical to the security. We observe that the vulnerable function `telrcv()` can be invoked after the initializations of `loginprg` and `host` variables, but before the invocation of `start_login(host)`. Therefore, the exploitation of the heap overflow vulnerability allows overwriting `loginprg` value to `/bin/sh`, and `host` value to `–p`, so that the command `/bin/sh –h –p –p` will be executed by function `start_login()`, giving a root shell to the

attacker. Note that if `host` was not overwritten or it was overwritten to an empty string, `sh` command would generate a *File Not Exist* error.

Table 4: Attacking *loginprg* and *host* variables in Telnet Daemon

```
void start_login(char * host,…) {
   addarg(&argv, loginprg);
   addarg(&arg, ”-h”);
   addarg(&argv, host);
   addarg(&arg, ”-p”);
   execv(loginprg, argv);
}
```

```
Without the corruption, the execv call is:
    /bin/login –h attacker.com –p

Due to the corruption, the execv call is:
    /bin/sh –h –p -p
```

4.4.3. Stack Buffer Overflow Attack against User Input Data

Another HTTP server, GHTTPD, has a stack buffer overflow vulnerability in its logging function [33]. Unlike the heap corruption, integer overflow or format string vulnerabilities, a stack overflow does not allow corrupting arbitrary memory locations, but only the memory locations following the unchecked buffer on the stack. The most popular method to exploit stack buffer overflow vulnerability is to use the stack-smashing method, which overwrites a return address [33]. The attack overwrites the function return address saved on stack, and changes it to the address of the injected malicious code, which is also saved in the unchecked buffer. When the function returns, it begins to execute the injected code. Stack buffer overflow attacks have been extensively studied and many runtime protection solutions have been proposed. Most of the techniques try to detect corruption of return addresses. We construct an attack that neither injects code nor alters the return address. The attack only alters the backup value of a register in the function frame of the vulnerable function to compromise the security validation checks and eventually cause the root compromise.

The stack buffer overflow vulnerability is in function `log()`, where a long user input string can overrun a 200-byte stack buffer. A natural way to conduct a non-control-data attack is to see if any local stack variable can be overwritten. We were not able to find any local variable that can be used to compromise its security. Instead, we found that three registers from the caller were saved on the stack at the entry of function `log()` and restored before it returns. Register `ESI` holds the value of the variable `ptr` of the caller

function `serveconnection()`. Variable `ptr` is a pointer to the text string of the URL requested by the remote client. Function `serveconnection()` checks if the substring "`/..`" (i.e., the parent directory) is embedded in the requested URL. Without the check, a client could execute `www.foo.com/cgi-bin/../bar`, an executable outside the restricted CGI-BIN directory. We observe that function `log()` is called after `serveconnection()` checks the absence of "`/..`" in the URL, but before the CGI request is parsed and handled. This makes a TOCTTOU (Time of Check To Time of Use) attack possible – we first present a legitimate URL without "`/..`" to bypass the absence check, then we change the value of register `ESI` (value of `ptr`) to point to a URL containing "`/..`" before the CGI request is processed.

Table 5: Source Code of `servconnection()` and `log()`

```
int serveconnection(int sockfd) {
char *ptr;  // points to the URL
            // ESI is allocated
            // to this variable
...
1: if (strstr(ptr,"/..”))
  reject the request;
2: log(...);
3: if (strstr(ptr,"cgi-bin”))
4:     Handle CGI request
...
}
```

```
Assembly of log(...)
push %ebp
mov %esp, %ebp
push %edi
push %esi
push %ebx
... stack buffer overflow code
pop %ebx
pop %esi
pop %edi
pop %ebp
ret
```

The attack scheme is given in Figure 13. The default configuration of CGI-BIN of GHTTPD is `/usr/local/ghttpd/cgi-bin`, so the path `/cgi-bin/../../../../bin/sh` is effectively the absolute path */bin/sh* on the server. We use the *GET* command of the HTTP protocol to trigger the buffer overflow condition and force the server to run */bin/sh* as a CGI program: we send the command "`GET AA…AA\xdc\xd7\xff\xbf↵↵/cgi-bin/../../../../bin/sh`" [13] to the server. The server converts the first part of the command, "`AAA…AAA\xdc\xd7\xff\xbf`", into a null-terminated string pointed to by `ptr` in function `serveconnection()`. This string passes the "`/..`" absence check in Line 1 of `serveconnection()`. When the string is passed to the `log()` function in Line 2, it overruns the buffer and change the saved copy of register `ESI` (i.e., `ptr`*)* on the stack frame of `log()` to `0xbfffd7dc` (i.e., the bytes following "`A`" characters in the request),

[13] "AA…AA" represents a long string of "A" characters.

which is the address of the second part of the GET command "`/cgi-bin/../../../../bin/sh`". When `log()` returns, the value of `ptr` points to this unchecked string, which is a CGI request containing "`/..`". Succeeding in the check of Line 3, the request eventually starts the execution of `/bin/sh` at Line 4 under the root privilege.

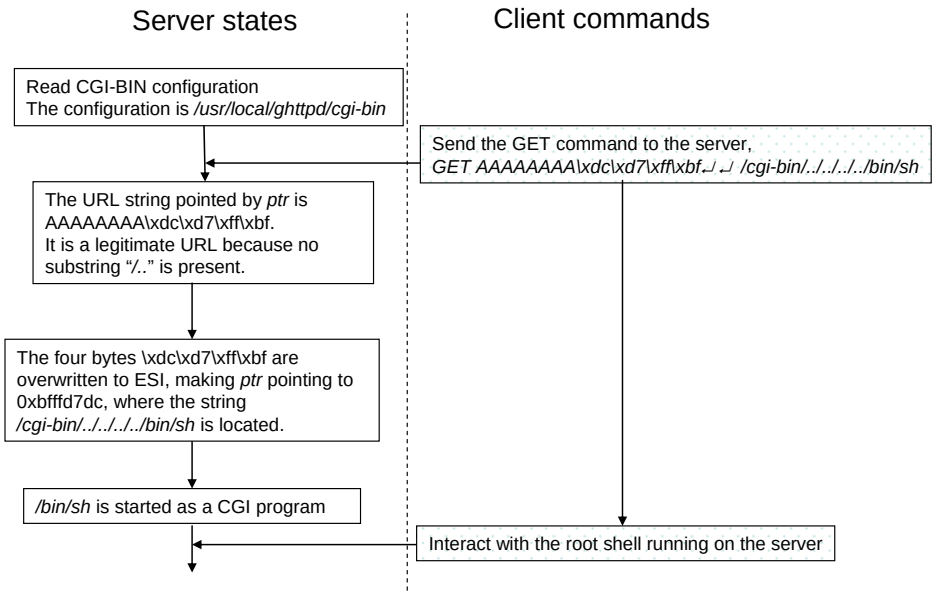


Figure 13: User Input Data Attack in GHTTPD

4.4.4. Integer Overflow Attack against Decision-Making Data

We also study decision-making data used by security-related operations in server applications. These data are usually Boolean variables used to see whether certain criteria are met by a remote client. If so, access will be granted. An attacker can exploit security vulnerabilities in a program to overwrite such Boolean variables and get access to the target system. We study the attack in the context of a secure shell (SSH) server implementation.

An integer overflow vulnerability [70] exists in multiple SSH server implementations, including one from *SSH Communications Inc.* and one from *OpenSSH.org*. The vulnerability is triggered when an extraordinarily large encrypted SSH packet is sent to the server. The server copies a 32-bit integer packet size value to a 16-bit integer. The 16 bit integer can be set to zero when the packet is large enough. Due to this condition, an arbitrary memory location can be overwritten by the attacker.

Available exploitation online changes a function return address to run malicious shell code [70]. Detailed descriptions and analyses of this vulnerability can be found in [58] and [72].

Our goal is to corrupt non-control data in order to login to the system as root without providing a correct password. We study the source code of the SSH server implementation from SSH Communications Inc. The integer overflow vulnerability is in function `detect_attack()`, which detects the *CRC32 compensation attack* against the SSH1 protocol. This function is invoked whenever an encrypted packet arrives, including the encrypted user password packet. The SSH server relies on function `do_authentication()` (shown in Table 6) to authenticate remote users. It uses a *while* loop (line 2) to authenticate a user based on various authentication mechanisms, including *Kerberos* and *password*. The authentication succeeds if it passes any one of the mechanisms. A stack variable `authenticated` is defined as a Boolean flag to indicate whether the user has passed one of the mechanisms. The initial value of *authenticated* is 0 (i.e., false). Line 3 reads input packet using `packet_read()`, which internally invokes the vulnerable function `detect_attack()`. Our attack is to corrupt the `authenticated` flag and force the program to break out of the `while` loop and go to line 9, where a shell is started for the authenticated user.

Table 6: Source Code of do_authentication()

```
void do_authentication(char *user, ...)
{
1:  int authenticated = 0;
...
2:  while (!authenticated) {
   /* Get a packet from the client */
3:    type = packet_read();
      // calls detect_attack()
internally
4:    switch (type) {
   ...
5:    case SSH_CMSG_AUTH_PASSWORD:
6:     if (auth_password(user,
password))
7:          authenticated =1;
      case ...
   }
8:    if (authenticated) break;
 }
 /* Perform session preparation. */
9:  do_authenticated(pw);
}
```

Our attack tries to login as root without providing a correct password: when the server is ready to accept the root password, the SSH client sends a very large packet to the receiving function `packet_read()` (Line 3). The packet is specially formulated to trigger the integer overflow vulnerability when `packet_read()` calls `detect_attack()` for detection. As a result, the `authenticated` flag is changed to non-zero. Although the server does fail in function `auth_password()` (Line 6), it breaks out of the `while` loop and proceeds to create a shell for the client (Line 9). The client program successfully gets into the system without providing any password. Figure 14 shows the status of both the client and the server during the attack.

Currently our attack program has not calculated the correct checksum of the malicious packet that we sent to the server, so the packet would be rejected by the checksum validation code in the SSH server. For a proof-of-concept attack, we deliberately make the server accept the malicious packet without validating its checksum. To make the attack complete, we need to understand the DES cryptographic algorithms to recalculate the checksum. Note that an attack including the checksum calculation algorithm is publicly available [72]. Other than this peculiarity, we have confirmed that the vulnerability allows precise corruption of `authenticated` flag, and that this corruption is sufficient to grant the root privilege to the attacker.

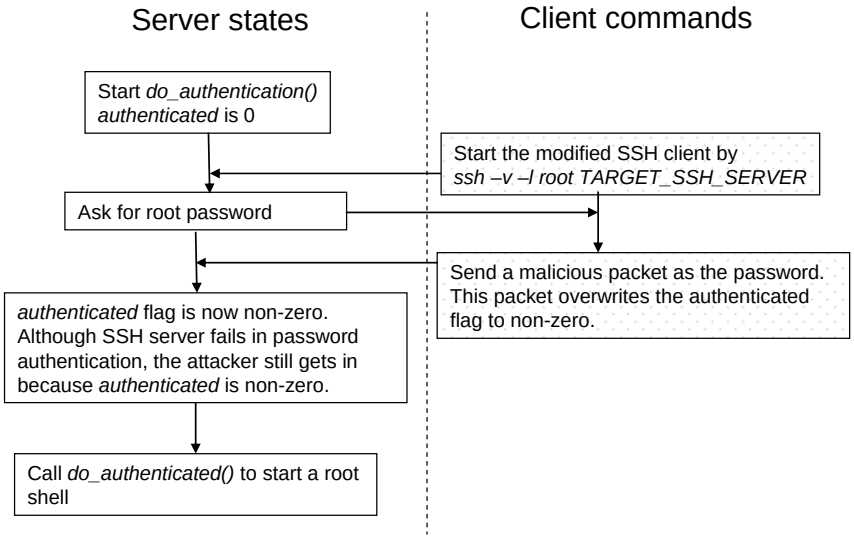


Figure 14: Attacking Stack Variable authenticated in SSH Server

4.5. Implications on Defensive Techniques

The success in constructing non-control-data attacks for various network server applications suggest a re-examination of many current defensive techniques, which can be broadly categorized into two classes: techniques to avoid having memory-safety bugs in software and techniques to defeat exploitations of these bugs. We discuss these techniques below and the impact of our result on them.

4.5.1. System Call Based Intrusion Detection Techniques

Many host-based instruction detection systems (IDS) monitor the behaviors of an application process at system call level. These systems build abstract models of a program based on system call traces. At runtime, the IDS monitor the system calls issued by the program. Any deviation from the pre-built abstract model is considered abnormal or incorrect behavior of a program. One of the earliest attempts was by Forrest et. al. [31] [37] in which they used short sequences of system calls (*N-grams*) obtained from training data to define a process's correct behavior. The monitoring is a matter of sequence matching against the pre-built *N-gram* database. Wagner and Dean [77] build abstract system call models from the control flow graph based on static source code analysis. A basic non-deterministic finite automaton (NDFA) and a more powerful non-deterministic pushdown automaton (NPDA) that incorporates stack state are built. Sekar et al. [63] improves Forrest's training method. They built a finite state automaton (FSA) constructed from training system traces by associating system calls with program counter information. Feng et. al. [29] further improves the training method in the VtPath model. At every system call, VtPath extracts the virtual stack list, which is the list of return addresses of functions in the call stack. Then a virtual path is extracted from two consecutive virtual stack lists and stored as a string in a hash table. VtPath detects some attacks that are missed by the FSA model. In a follow-up paper, Feng et. al. [30] proposed a static version of VtPath, called VPStatic, and compared to the DYCK by Griffin et. al. [34] which constructs PDA models directly from binary code. Gao et al. [32] proposes the execution graph model that uses training system call traces to approximate the control flow graph that is usually only available through static code analysis. The execution graph is built by considering both program counter and call stack information.

All these intrusion detection methods monitor process behaviors at the system-call level, that is, they are only triggered upon system calls. As shown earlier, non-control-data attacks require no invocation of system call, therefore the attacks will most likely evade the detection of the system-call based monitoring mechanisms. Data flow information needs to be incorporated in these IDS models in order to detect non-control-data attacks.

Some IDS techniques [41] abstract program normal behaviors using statistical distributions of system call parameters. The distribution is obtained from training data. At runtime, the IDS detects program anomalies by observing deviation from the training model. These methods detect intrusions based on the anomalies of data rather than the anomalies in control flow. Therefore, we believe that with proper training, they can detect some of the non-control-data attacks: the HTTPD CGI-BIN attack when */bin/sh* is run by the *execve()* since that is most likely not in the training model. The method, however, is not able to detect the decision-making data attack in Section 4.4.4 where no system call parameter is modified; it cannot detect the user-identity data attack discussed in Section 4.4.1 without considering control flow information in the training model. It might be difficult for a statistical algorithm to precisely extract a fine-grained policy to detect the attacks with a reasonably low false positive rate. Despite such technical difficulties, considering system call parameter anomalies is one possible way to extend current IDS's in detecting some non-control-data attacks.

4.5.2. Control Data Protection Techniques

Corrupting control data to alter the control flow is a critical step in traditional attacks. Compiler techniques and processor architecture level techniques have been proposed in very recent papers to protect control data. DIRA is a compiler to automatically insert code only to check the integrity of control data [68]. An explicitly stated justification of this technique is that control flow diversion attacks are currently considered the most dominant attacks. Suh, Lee and Devadas developed the *Secure Program Execution* technique to defeat memory corruption attacks [73]. The idea is to tag the data directly or indirectly derived from I/O as *spurious* data, which a concept more commonly referred to as tainted data in other literatures [28][59][66]. Security attacks are detected when tainted

data is used as an instruction or jump target addresses. Another recent work on control data protection is *Minos* [26], which extends each memory word with an *integrity* bit. *Integrity* indicates whether the data originating from a trusted source. It is essentially the negation of *taintedness*. Very similar to *Secure Program Execution*, *Minos* detects attacks when the integrity bit of a control data is 0.

We agree that control data are highly critical in security-related applications. Not protecting them allows attacks to easily succeed. While the authors of the above defensive techniques assume that protecting control data integrity can defeat most attacks exploiting memory corruption vulnerabilities, this assumption may be unsafe when the general applicability of non-control-data attacks is considered.

4.5.3. Non-Executable Memory Based Protections

A number of defensive techniques are based on non-executable memory pages, which block the attackers' attempt to inject malicious code onto a writable memory page and later divert program control to execute the injected code. StackPatch is a Linux patch to disallow executing code on the stack [69]. Microsoft has also implemented non-executable memory page supports in Windows XP Service Pack 2 [3]. In addition, the latest versions of Linux and OpenBSD are enhanced with similar protections.

These defensive techniques cannot defeat non-control-data attacks, because there is no attempt to run any injected code during the attacks. Note that non-executable memory based protections can also be defeated by the *return-to-library* attacks, which divert program control to library code, instead of the injected code [79].

4.5.4. Memory Safety Enforcement

CCured [52] is a program transformation tool that attempts to statically verify that a C program is type-safe, and thus free from memory errors. When static analysis is insufficient to prove type-safety, it instruments vulnerable portions of code with checks to avoid errors such as NULL pointer dereferences, out-of-bounds memory accesses, and unsafe type casts. Its main mechanism for enforcing memory safety is a type-inference algorithm that distinguishes pointers by how safely they are used. Based on this classification of pointers, code transformations are applied to include appropriate runtime

checks for each type of pointer. Although *CCured*'s analysis techniques and runtime system are sophisticated and guarantee memory safety, instrumenting existing programs often require nontrivial source code changes to ensure compatibility with external libraries.

CRED [61] is a buffer overflow detector that uses the notion of referent objects to add bounds checking to C without restricting safe pointer. Any addresses resulting from arithmetic on a pointer must lie within the same memory object as that of the pointer. To enforce this, *CRED* stores the base address and size of all memory objects in the object table. Immediately before an in-bounds pointer is used in an arithmetic operation, its referent object's bounds data is retrieved from the object table. This data is used to ensure the pointer arithmetic's result lies within the bounds of the referent object. When an out-of-bounds address is used in pointer arithmetic, its associated referent object's bounds data is used to determine if the resulting address is in bounds. To reduce performance overhead, *CRED* limits its bounds checking to string buffers, which implies that *CRED* does not provide protection against attacks involving non-string buffers. In addition, programs that perform heavy string-processing (e.g., web/email servers) can still incur overheads as high as 200%.

Cyclone [40] is a memory-safe dialect of C that aims to maintain much of C's flexible, low-level nature. It ensures safety in C by imposing a number of restrictions. Like *CCured*, *Cyclone* adds several pointer types that indicate how a pointer is used and inserts appropriate runtime checks based on a pointer's type. Porting C programs to *Cyclone*, however, can be difficult due to its additional restrictions and semantics. For example, *Cyclone* only infers pointer kinds for strings and arrays. As such, it is often the programmer's responsibility to determine the appropriate type for a pointer. This task can be very time-consuming for large programs that make extensive use of pointers. In addition, *Cyclone* programs often perform significantly worse than their C counterparts and commonly-used software development tools such as compilers and debuggers must be modified for use with *Cyclone* source code.

SAFECode [27] is a compiler technique with the ultimate goal of enabling 100% static memory-safety enforcement for programs running on embedded systems. The key technique is an automatic pool allocation algorithm. Although *SAFECode* does impose

certain language restrictions on C programs, it has been tested against a diverse set of embedded C programs. Because it is a static compiler analysis technique, it does not incur the runtime overhead as *CCured* and *Cyclone*.

Although the techniques enforcing memory-safety continue to show great promise, the software engineering community has not established techniques that allow an easy migration path from current large code bases, and thus memory-safety bugs are likely to still exist for an extended period of time. For this reason, research efforts should still be invested in defensive techniques that assume the existence of memory-safety bugs.

4.5.5. Other Defensive Techniques

We now discuss other runtime defensive techniques which do not assume the control-data attack pattern.

Specialized techniques of which the effectiveness is not affected by non-control-data attacks. Some specialized dynamic detection techniques are not affected by non-control-data attacks: *StackGuard* [25] and *Libsafe* [7] can still defeat many stack buffer overflow attacks unless security sensitive data are in the same frame as the overflowing buffer, like the one in GHTTPD example. *FormatGuard* [23] is still effective to defeat format string attacks because it does not allow overwriting arbitrary memory addresses. However, these techniques are not generic enough to defeat attacks exploiting other types of vulnerabilities.

*Generic techniques requiring improvements***.** Among various techniques that address a broader range of memory vulnerabilities, the underlying principles of the pointer protection technique *PointGuard* [24], address-space randomization techniques [9][57] and TaintCheck [54] are sound, but improvements are needed to better deploy these principles:

PointGuard is a compiler technique that embeds pointer encryption/decryption code to protect pointer integrity in order to defeat most memory corruption attacks. In principle, if all pointer, including pointers in application code and in libraries (e.g., LibC), are encrypted, most memory corruption attacks can be defeated. However, without the instrumented library code, the current PointGuard cannot defeat many non-control-data attacks. For example, the previous presented heap overflow and format string attacks only

corrupt heap free-chunk pointers and the argument pointers of *printf*-like functions, which are pointers in LibC. Although there are technical challenges in the instrumentation of PointGuard at the library level, such as the lack of accurate type information, we argue that such an improvement is essential.

The principle of *address-space randomization* techniques is to rearrange memory layout so that the actual addresses of program data are different in each execution of the program. Ideally, the addresses should be completely unpredictable. Nevertheless, Shacham et al. [65] has recently shown that most current randomization implementations on 32-bit architectures suffer from the low entropy problem – even with very aggressive re-randomization measures, these techniques cannot provide more than 16-20 bits of entropy, which is not sufficient to defeat determined intruders. Deploying address-space randomization techniques on 64-bit machines is considered more secure.

TaintCheck [54] uses a software emulator to track the taintedness of application data. Depending on its configuration and policies, TaintCheck can perform checks on a variety of program behaviors, e.g., use of tainted data as jump target, use of tainted data as format string, and use of tainted data as system call arguments. Preventing the use of tainted data as system call arguments can be used to detect some, but not all non-control-data attacks. However, as the authors of TaintCheck have pointed out, this can lead to false positives as some applications do require legitimately embedding tainted data in the system call arguments. Further, the reported runtime slowdown is between 5-37 times. Further research is required to address the issues of security coverage, false positive rate and runtime overhead.

4.5.6. Defeating Memory Corruption Attacks: Still Challenging in Practice

The above analysis shows that finding a generic and secure runtime technique to defeat memory corruption attacks is still an open problem. The specialized techniques can only defeat attacks exploiting a subset of memory vulnerabilities. For generic defensive techniques, many of them provide security by enforcing control flow integrity, and thus the security coverage is incomplete due to the general applicability of non-control-data attacks. A few other generic solutions, although not fundamentally relying

on control flow integrity, need improvements to overcome the practical constraints in their deployments.

4.6. Empirical Discussions on Mitigating Factors

Despite the general applicability of non-control-data attacks, we did experience more difficulties when constructing these attacks as compared to constructing control-data attacks. In particular, the requirement of application-specific semantic knowledge and the lifetime of security critical data are major mitigating factors that impose such difficulties on attackers.

4.6.1. Requirement of Application-Specific Semantic Knowledge

An obvious constraint for constructing non-control-data attacks is attakcers' reliance on application-specific knowledge. In a control-data attack, as long as a function pointer or a return address can be overwritten, a generic piece of shell code will be started to do all kinds of security damages easily. However, a non-control-data attack must preserve control flow integrity, so an attacker needs to have in-depth knowledge about how the target application behaves. For example, to attack HTTP servers, we need the insights into the CGI mechanism; to attack WU-FTP server, we should know how the effective UID is elevated and dropped. In the current stage, we have not formulated an automatic method to obtain such knowledge. The method that we used in attack constructions is a combination of vulnerability report review, debugger-aided source code review and certain diagnostic tools such as *strace* (system call tracer) and *ltrace* (library call tracer). This is apparently a time-consuming process.

However, we argue that this is not a fundamental constraint on attackers because, 1) knowledge of widely used applications is not hard to obtain, so a determined attacker is likely to eventually succeed no matter how long it takes, if there is a strong incentive; 2) Although we spent lots of effort to construct these attacks, future attackers may not need the same amount of effort. For example, suppose a new vulnerability was found in another HTTP server, an attacker would easily think of attacking CGI-BIN configuration. This can be similar to the history of the stack-smashing attack – it was a mystery when the Morris Worm spread, but is now straightforward to understand.

4.6.2. Lifetime of Security Critical Data

Lifetime of security critical data is another constraint on seeking non-control-data attacks. The lifetime of a value is defined as the interval from the time when the value is stored in a variable to the time of its last reference before the variable is de-allocated or reassigned. Only when a vulnerability is exploitable during the lifetime of some security critical data can an attack succeed.

Our experience shows that although there are many potential data critical to security, a majority of them are eliminated by the constraints of value lifetime – the vulnerability occurs either before the data value is initialized or after the semantically diverging operation is performed. Therefore, reducing the lifetime should be considered as a secure programming practice. Two of the discussed attacks would not succeed if the programs were slightly changed as shown in Table 7. The original WU-FTPD function `getdatasock()` uses the global data `pw->pw_uid` in the `seteuid` call, allowing any vulnerability occurring before `getdatasock()` to escalate the process privilege. If the function was written as (A2), where a short-living local variable is used, only a vulnerability occurring within the lifetime of `tmp` (denoted as an arrow) could affect the `seteuid` call. Similarly, in the original SSHD `do_authentication()` (code B1), the lifetime of the `authenticated` value covers the vulnerable `packet_read()` call. By inserting the statement "`authenticated=0`" after Line L1 in code B2, `authenticated` flag is always refreshed in every iteration, and thus its lifetime becomes shorter. The attack could not succeed since the vulnerability in L1 was out of the lifetime of `authenticated` flag.

Table 7: Reducing Data Lifetime for Security

(A1) Original WU-FTPD getdatasock()

```
{ seteuid(0);
  setsockopt( ... );
  seteuid(pw->pw_uid);
}
```

(A2) Modified WU-FTPD getdatasock()

```
{ tmp = geteuid();
  seteuid(0);
  setsockopt( ... );
  seteuid(tmp);
}
```

(B1)Original SSHD do_authentication()

```
{ int authenticated = 0;
  while (!authenticated) {
L1:type = packet_read(); //vulnerable
   switch (type) {
    case SSH_CMSG_AUTH_PASSWORD:
     if (auth_password(user, passwd))
          authenticated = 1;
    case ...
   }
   if (authenticated) break;
  }
  do_authenticated(pw);
}
```

(B2)Modified SSHD do_authentication()

```
{ int authenticated = 0;
  while (!authenticated) {
L1:type = packet_read(); //vulnerable
   authenticated = 0;
   switch (type) {
    case SSH_CMSG_AUTH_PASSWORD:
     if (auth_password(user, passwd))
          authenticated = 1;
    case ...
   }
   if (authenticated) break;
  }
  do_authenticated(pw);
}
```

The lifetimes of security critical configuration data, as those in the NULL HTTPD attack and the Telnetd attack, are more difficult to reduce. A possible protection solution is to encrypt them in a similar way as the encryption technique used by *PointGuard* or to set the memory of configuration data read-only.

4.7. Summary of Non-Control-Data Attack Applicability

We begin with the applicability claim that *many real-world software applications are susceptible to attacks that do not hijack program control flow, and the severity of the resulting security compromises is equivalent to that due to control-data attacks.* The claim is empirically validated by the experiments on constructing non-control-data attacks against many major network server applications. Each attack exploits a different type of memory vulnerability to corrupt non-control data and obtain the privilege of the victim process. The results of the experiments indicate that control flow integrity may not be a sufficiently accurate approximation to software security. The general applicability of non-control-data attacks represents a realistic threat to be considered seriously in defense research.

We study a wide range of current defensive techniques and discuss how the general applicability of non-control-data attacks affects the effectiveness of these techniques. The analysis shows the necessity for further research on defenses against memory corruption

based attacks. Finding a generic and secure solution to defeat memory corruption attacks is still an open problem.

Despite their general applicability, non-control-data attacks are less straightforward to construct compared to control-data attacks because they require insight semantic knowledge about the target application. Another important constraint is the lifetime of security-critical data. We suggest that reducing data lifetime is a secure programming practice that will increase software resilience to attacks.

CHAPTER 5

REASONING ABOUT VULNERABILITIES USING POINTER TAINTEDNESS SEMANTICS

5.1. Pointer Taintedness

The FSM analysis work presented in CHAPTER 3 and the study of non-control-data attacks presented in CHAPTER 4 prompt us to seek defensive technique providing better security protection coverage via identifying a root cause of most memory corruption attacks. The FSMs of stack buffer overflow, heap corruption, integer overflow and format string vulnerability presented earlier suggest the existence of a common root cause – *pointer taintedness*, which refers to the program behavior that during the execution of a program, a pointer value (including return address) is derived directly or indirectly from user input. Since pointers are internal to applications, they should be transparent to users. Thus a taintable pointer suggests a potential security vulnerability.

In order to support the claim that pointer taintedness is indeed the root cause of a wide range of real vulnerabilities, this section gives detailed illustrations about the internals of format string vulnerability, heap corruption vulnerability, stack buffer overflow and *globbing* vulnerability. We then propose a theorem-proving based source code analysis technique to reason about these security vulnerabilities based on the notion of pointer taintedness.

5.1.1. Format String Vulnerability

Format string vulnerability is caused by incorrect invocation of `printf`-like functions (e.g., `printf, sprintf, snprintf, fprintf` and `syslog`). Table 8 gives examples of correct and incorrect invocations of `Printf()` (our simplified version of LibC `printf()` that we developed). This sample program produces five lines of output (shown in Table 8). Output lines O1 and O2 result from executing line L3 of the code: the string *buf* hosting "*hello*", and the integer `i` has the value 1234. In addition, line L3 uses the format directive `%n` to write the character count (i.e., the

number of characters printed up to that point) into the address of the corresponding integer variable. In this case, the length of "`string=hello↵data=1234↵`" is 23, so `Printf()` writes 23 to the integer `j`. This value is printed out in line O3 of the program output. The format string vulnerability is caused by incorrect invocation of `Printf` in line L4, which directly uses `buf` as the format string (the proper usage should be `Printf("%s", buf)`).

This vulnerability is usually exploited in the following manner. Let's assume that the attacker wants to corrupt an arbitrary memory location (e.g., the global integer *i*). In order to do this, he/she constructs an input string `buf` as given below (observe that the beginning of the input string corresponds to the address of global integer *i*):

\x78 \x99 \x04 \x08 %d %d %d '1' '2' '3' '4' '5' %n

The string is read by `scanf()` and passed to `Printf()`, which in turn, calls `Vfprintf()`. Just before Line L1 is executed, the stack layout is like the one in Figure 15. In `Vfprintf()`, there are two pointers: `p` is the pointer to sweep over the format string `buf` (from "\x78" to"%n"), and `ap` is the pointer to sweep over the arguments (starting from the 12-byte gap). The attacker deliberately embeds three "%d" directives in `buf` so that `ap` can consume the 12-byte gap and get to the word `0x08049978`. A padding string "12345" follows the "%d" directives in order to adjust the character count. As we see in the program output line O4, the words in the 12-byte gap are printed as three integers followed by a padding string "12345". Eventually, when `p` arrives at the position of "%n" (i.e., the code line L1 is about to be executed), `ap` happens to arrive at the position of `0x08049978`. Line L1 writes the character count `count` to the location pointed by `*ap`. In this case, since the content in the location pointed by `*ap` is the address of the integer `i`, the character count 31 is written to `i`. Note that this attack can overwrite any memory location, including locations containing return addresses or the global offset table of an application, which can result in the execution of the attacker's code.

Table 8: Format String Vulnerability Illustration

```
   int Vfprintf (FILE *s, const char *format, va_list ap)
   {      char * p;
         int count;
           p = format; … …
L1:     *(int *) ap = count;
           … …
   }

   int Printf (const char *format, ...)
   {   va_list arg;     … …
L2:    Vfprintf (stdout, format, arg);
      … …
   }

   int i,j;
   int main()
   {  char buf[100];
      //This is how to call Printf correctly
      strcpy(buf,"hello");
      i=1234;
L3:   Printf("string=%s\ndata=%d\n%n",buf,i,&j);
      Printf("total output length=%d\n",j);

       //This is how format string vulnerability occurs
      scanf("%s",buf);
 L4:  Printf(buf);
 L5:  Printf("\ni=%d\n",i);
   }

Program Output:
O1:         string=hello
O2:         data=1234
O3:         total output length=23
O4:             134514747123413451916812345
O5:         i=31
```

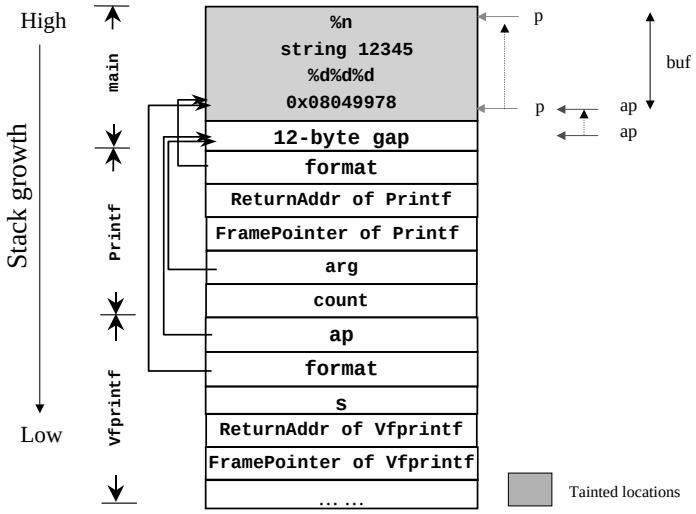


Figure 15: How to Overwrite the Global Integer *i*

We can view the above vulnerability as a consequence of pointer taintedness. In the above code (Table 8), the string `buf` is obtained from user input and is hence tainted (as indicated in Figure 15 as a grey area). When the pointer `ap` sweeps over the stack and

points to `buf`, `*ap` becomes tainted. `ap` is then dereferenced in Line L1, and the tainted value of `*ap` is the target address of the write operation. This can lead to the corruption of an arbitrary memory location. Thus we see that pointer taintedness is the root cause of this vulnerability. Note that the pointer `*ap` gets tainted because `ap` moves into the tainted memory locations, and there is no explicit assignment of a tainted value to `*ap` in the C code. Hence a memory model is necessary to reason about the taintedness of `*ap`. The next section defines the formal semantics of pointer taintedness using a memory model, and Section 5.3 shows how the semantics can be used to reason about security vulnerabilities in library functions.

5.1.2. Heap Corruption Vulnerability

Heap corruption vulnerabilities (including heap buffer overflow and double free) may result in modifying arbitrary memory locations and execution of an arbitrary code. To illustrate heap corruption vulnerabilities, we ported an implementation of binary buddy heap management system (originally implemented on Windows) to Linux.

Table 9: Heap Corruption Vulnerability Illustration

```
     typedef struct _HEAP_BLOCK {
         int             Size;           // The size of the block.
         int             Busy;           // Is this block busy?
         struct _HEAP_BLOCK * Fwd,* Bak;  // List to the free blocks
                                          //     of same size
     } HEAP_BLOCK, * PHEAP_BLOCK;
     void Free(PVOID p)
     {   HEAP_BLOCK * BuddyBlock;
L0       … …
         BuddyBlock->Fwd->Bak=BuddyBlock->Bak ;
         … …
     }
     void foo() {
     }
     int main()
L1   {   char * p;
         void (*f)();
L2       f=foo;
L3       p = Malloc(40);
L4       printf("At the begining, f=%p p=%p\n",f,p);
         *(UINT*)(p+60)=(UINT)p;
         *(UINT*)(p+56)=((UINT)&f)-12;
         Free(p);
         printf("\nAfter Free, f and p are identical:");
         printf("f=%p p=%p\n",f,p);
     }
     Program output:
     At the begining, f=0x80488cc p=0x8049170
     After Free, f and p are identical: f=0x8049170 p=0x8049170
```

The implemented heap management code consists of two functions `Malloc()` and `Free()`[14] functionally similar to LibC `malloc()` and `free()`. Table 9 illustrates how to make a function pointer `f` pointing to a buffer `p` by corrupting the heap structure. Line L1 allocates a 40-byte buffer to p. Suppose the locations of (`p+56`) and (`p+60`) are tainted by attackers due to buffer overrun or double free vulnerabilities, they can be corrupted in such a way that is similar to the effect of Lines L2 and L3 (in real attack scenarios, L2 and L3 are performed in attacker's programs). The buddy heap system organizes unallocated memory blocks as several double-linked lists called free-chunk lists. Figure 16 shows an example list of free-chunks. In the implementation, the header of free-chunks is defined as `HEAP_BLOCK` structure. Figure 16 shows four `HEAP_BLOCKs`. `HEAP_BLOCK` 1, 3 and 4 form the free-chunk double-linked list. (`p+56`) is the location of the forward link (`Fwd`) of `p`'s buddy block, and (`p+60`) is the location of the backward link (`Bak`) of `p`'s buddy block. We use `FreedBlock` to represent the heap block containing buffer `p`. When the buffer `p` is freed by the function `Free()`, `FreedBlock`'s buddy block should be removed from its free-chunk double-linked list and merged with `FreedBlock` to form a larger free-chunk. The removal of `FreedBlock`'s buddy block is performed in Line L0, which reads `BuddyBlock->Fwd->Bak=BuddyBlock->Bak`. However, because `BuddyBlock->Fwd` and `BuddyBlock->Bak` are corrupted in Lines L2 and L3, the effect of executing Line L0 becomes `*((((UINT)&f)-12)+12)=(UINT)p`, which is equivalent to `f=p` (Note that "12" is the offset of the `Bak` field in the `HEAP_BLOCK` structure). As we see in the second line of the output, after `Free()` is called, the function pointer f now points to the buffer `p`. If the buffer `p` contains attacker's malicious code, calling function pointer `f` later will execute the malicious code. In this example, we again observe the scenario of tainted pointers (i.e., `BuddyBlock->Fwd` and `BuddyBlock->Bak`) being de-referenced in Line L0.

[14] The source code of `Malloc()` and `Free()` are given in Appendix B.

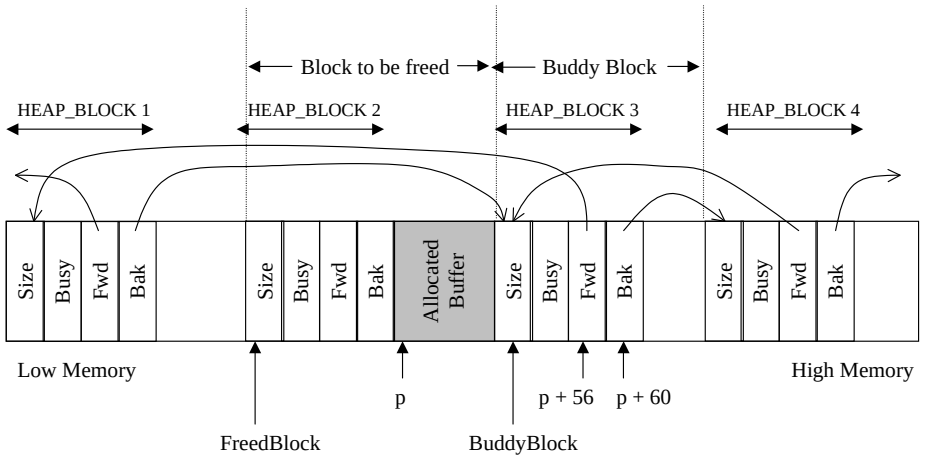


Figure 16: Normal Heap Structure before Buffer p is freed

5.1.3. Stack Buffer Overflow

Stack buffer overflow vulnerabilities are most well-known vulnerabilities. We also constructed a proof-of-concept program for this type of vulnerability, which aims to overwrite the return address by copying a user-input string to a buffer on the stack. In other words, return addresses are tainted by user input. Due to space limitation, we do not illustrate this well-known vulnerability category here.

5.1.4. `Glob()` Vulnerabilities

Globbing is the UNIX filename matching functionality. It matches filenames with a pattern containing wildcards. For example, when a user issues an UNIX command ls `~/*.txt`, all filenames ending with .txt on the user's home directory are listed.[15] We found five entries in *Bugtraq* due to globbing vulnerabilities. *Bugtraq* ID: 2548, *Bugtraq* ID: 2550 and *Bugtraq* ID: 2552 are buffer overflow vulnerabilities. The causes of these vulnerabilities are discussed in previous sections.

The causes of *Bugtraq* ID: 3581 (*Wu-Ftpd File Globbing Heap Corruption Vulnerability*) and *Bugtraq* ID: 3707 (*Glibc File Globbing Heap Corruption Vulnerability*) are not buffer overflow, but both vulnerability descriptions also show the existence of pointer taintedness situations. Table 10 is compiled from the vulnerability description of *Bugtraq* ID: 3707. We see that when the four characters `\xef\xef\xbe\xad\xde` are embedded in the input, the values of registers `ESI` and

[15] Use UNIX command "man 7 glob" to see the full description of *globbing* functionality.

edi become `0xdeadbeef`. Eventually this data is used as the pointer to be freed, which is another instance of pointer taintedness. Note that the description of *Bugtraq* ID: 3581 is similar to this scenario.

Table 10: Glibc `Glob()` Vulnerability Description

```
Attacker's interaction with FTP server
   -> PASS AAAAAAAAAAAAAAAAAA\xef\xef\xbe\xad\xde # ( <19 Bytes> <Addr
towrite> <Glob char>)
   : 230 Guest login ok, access restrictions apply.
   -> STAT ~AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA{

Consequence of the attack
  Examination of the registers shows that we have successfully inserted the
intended address.
           esi            0xdeadbeef
           edi            0xdeadbeef
  On giving the ftp daemon a valid address to free, the daemon will continue to
free() the address we gave it.
```

5.2. Semantics for Pointer Taintedness

Starting with the programming semantics of Goguen and Malcolm [35], this section proposes a formal semantics to reason about pointer taintedness in programs. The semantics proposed in [35] defines instructions, variables and expressions. We extend this semantics to include memory locations and addresses. Using the memory model, the notion of taintedness is incorporated into the semantics.

We define tainted data as: (1) data coming from input devices (e.g., by `scanf(), fscanf(), recv(), recvfrom()`), or (2) data copied or arithmetically calculated from tainted data[16]. A tainted pointer is a pointer whose value (semantically equivalent to "data") is tainted. This definition can be formalized in equational logic using the Maude tool [22], which we used to reason about pointer taintedness.

In the semantics defined in [35], a `Store` represents the current state of all program variables. We extend this definition of a `Store` to be a snapshot of the entire memory state at a point in the program execution. The execution of a program instruction is defined as a function taking two arguments, a `Store` and an instruction, and producing another `Store`. There are two attributes associated with every memory location of a `Store`: *content* and *taintedness*. Accordingly, two operations, *fetch* and *location-*

[16] Taintedness is only propagated by arithmetic operations, such as +, * and arithmetic shifts. As a counter example, p is not a tainted value after the following C statement: `if (taintedValue>0) p=&fun1(); else p=&fun2();`

taintedness, are formally defined. The *fetch* operation `Ftch(S,I)` gives the content of the address `I` in store `S`; the *location-taintedness* operation `LocT(S,I)` returns a Boolean value indicating whether the content of the specified address is tainted.

There is no notion of "variable" in this semantics. Any variable in a C program is mapped to a memory location addressed by the integer with the same name as the program variable. For example, the C program variable `foo` is mapped as a memory location addressed by the integer `foo`. We define the `^` operator to dereference an integer, i.e., to fetch the location addressed by the integer. Note that the address (a.k.a., the left value) of the C program variable foo is represented by the integer foo in the semantics; and the content (a.k.a., the right value) of the C program variable `foo` is represented by (`^ foo`). The expressions in the semantics are arithmetic operators (e.g., +, - and *) concatenating integers and integer dereferences. For example, expression `200+(^foo)` represents "200 plus the content of the C program variable `foo`". Expression `200+foo` represents "200 plus the address of the C program variable `foo`".

We define two operations – *evaluation* and *expression-taintedness* – for expressions based on the *fetch* and *location-taintedness* operations. The *evaluation* operation `Eval(S,E)` gives the result of evaluating the expression `E` under store `S`; the *expression-taintedness* operation `ExpT(S,E)` indicates whether expression `E` contains any data from a tainted location, e.g., `ExpT(S,(^foo)+2)` indicates whether the expression `(^foo)+2` contains any data from a tainted location, which is equivalent to checking whether the memory location addressed by foo is tainted. *Thus pointer taintedness is defined as a dereference of a tainted expression.*

Table 11 lists a set of axioms for the *evaluation* and *expression-taintedness* operations, and gives examples of applying the equations for deduction. Lines 1-6 define how to evaluate an expression under store `S`. For example, line 1 indicates that the evaluation result of a constant `I` under store `S` is the constant `I`. Line 2 indicates that `Eval(S,^E1)` can be computed by first evaluating `E1` under `S`, then applying *fetch* operation on the evaluation result. The semantics of arithmetic operations are defined in Lines 3-6.

Table 11: Axioms of *Evaluation* and *Expression-Taintedness* Operations

	Axioms	Examples
1	Eval(S,I) = I	Eval(S,5) = 5
2	Eval(S,^E1) = Ftch(S,Eval(S,E1))	Eval(S,^foo) = Ftch(S,Eval(S,foo)) = Ftch(S,foo)
3	Eval(S,-E1) = -Eval(S,E1)	Eval(S, - 30) = - Eval(S, 30) = - 30
4	Eval(S,E1–E2) = Eval(S,E1)-Eval(S,E2)	Eval(S, 3-2) = Eval(S, 3) – Eval(S, 2) = 3 – 2 = 1
5	Eval(S,E1+E2) = Eval(S,E1)+Eval(S,E2)	Eval(S, 3+2) = Eval(S, 3) + Eval(S, 2) = 3 + 2 = 5
6	Eval(S,E1*E2) = Eval(S,E1)*Eval(S, E2)	Eval(S, 3*2) = Eval(S, 3) * Eval(S, 2) = 3 * 2 = 6
7	ExpT (S,I) = false	ExpT(S,5) = false
8	ExpT(S,^E1) = LocT(S,Eval(S,E1))	ExpT(S,^foo) = LocT(S,Eval(S,foo)) =LocT(S,foo)
9	ExpT(S,-E1) = ExpT(S,E1)	ExpT(S, -5) = ExpT(S, 5)= false
10	ExpT(S,E1-E2) = ExpT(S,E1) or ExpT(S,E2)	ExpT(S,(^foo)–2)=ExpT(S,(^foo))orExpT(S, 2) = LocT(S,foo) or false = LocT(S,foo)
11	ExpT(S,E1+E2) = ExpT(S,E1) or ExpT(S,E2)	ExpT(S,(^foo)+2)=ExpT(S,(^foo))or ExpT(S, 2) = LocT(S,foo) or false = LocT(S,foo)
12	ExpT(S,E1*E2) = ExpT(S,E1) or ExpT(S,E2)	ExpT(S,(^foo)*2)=ExpT(S,(^foo))or ExpT(S, 2) = LocT(S,foo) or false = LocT(S,foo)

Lines 7-12 define the *expression-taintedness* operator. Note that the relationship between the *expression-taintedness* and *location-taintedness* operators is similar to the relationship between the *evaluation* and *fetch* operators. Line 7 indicates that an integer constant is not a tainted expression. Line 8 indicates that determining whether the expression `^E1` is tainted is equivalent to checking whether the location addressed by the evaluation result of `E1` is tainted. Line 11 indicates that the expression E1+E2 is tainted if either `E1` or `E2` is tainted. The example of Line 12 shows that the expression `(^foo) + 2` is tainted if and only if the location pointed to by `foo` is tainted, according to the equation in Lines 7 and 8.

Table 12: Semantics of Statements

Statement	Semantics
`mov [E1] <- E2`	Move the evaluation result of the expression `E2` to the memory location addressed by the evaluation result of the expression `E1`
`if T then P1 else P2 fi`	If the condition `T` is true, Execute `P1` otherwise execute `P2`
`while T do P od`	If the condition `T` is true, execute `P`, repeat until `T` is false

Table 12 gives the informal semantics of a subset of the supported statements. Their formal semantics are similar to the specifications given in [35], and are sufficient to analyze a wide variety of program constructs in the C language. However, they are not sufficient to faithfully model all C statements. For example, the program counter has not been defined in the semantics. So certain C statements, such as `goto`, `break`, `continue`, `return` and `exit` cannot be modeled, but it is relatively easy to extend the semantics for these also.

Formal specifications of statements other than the mov statement are fairly straightforward. Axioms defining mov statement semantics are shown in Table 13. The goal is to define the *fetch* and *location-taintedness* operations after applying a mov instruction on store S. The semicolon operator in our notation represents the execution of an instruction on a store, which results in a new store. For example, `(S ; mov [E1]<-E2)` is the store after executing `mov [E1]<-E2` on store S. Line 1 indicates that if the expression `E1` evaluates to `X1` under store S, then when fetching the location `X1` after executing the instruction `mov [E1]<-E2`, we get the evaluation result of `E2` under store S. Line 2 indicates that if the expression `E1` does not evaluate to `X1` under S, then when fetching the location `X1` after executing the instruction `mov [E1]<-E2`, we still get the content in the location `X1` under S (i.e., before executing the instruction `mov [E1]<-E2`). Similarly, the *location-taintedness* semantics are defined for mov statement in Lines 3 and 4.

Table 13: Equations Defining `mov` Statement Semantics

1	`Ftch((S; mov [E1]<-E2),X1) = Eval(S,E2) if (Eval(S,E1) is X1) .`
2	`Ftch((S; mov [E1]<-E2),X1) = Ftch(S,X1) if not (Eval(S,E1) is X1) .`
3	`LocT((S; mov [E1]<-E2),X1) = ExpT(S,E2) if (Eval(S,E1) is X1) .`
4	`LocT((S; mov [E1]<-E2),X1) = LocT(S,X1) if not (Eval(S,E1) is X1) .`

5.3. Formal Reasoning of Pointer Taintedness Violations

This section performs pointer taintedness analysis for three common library functions based on the defined semantics, and extracts their associated security preconditions. The analysis identifies several well-known vulnerabilities, such as format string, buffer overflow and heap corruption vulnerabilities, thereby showing that pointer taintedness based reasoning is able to unify different kinds of vulnerabilities.

Our experience suggests that statements needing critical examination for pointer taintedness are typically indirect writes, where a pointer points to a target address to be written, e.g., the pointer `p` in `*p = foo` and `memcpy(p,foo,10)`. Checking indirect write statements is important because these statements can result in two types of pointer taintedness violations. For example, in the statement `*p = foo`, (1) if the value of `p` is tainted, then data foo can be written to any memory location; (2) if `p` is a pointer of a buffer but points to a location outside the buffer, then the statement `*p = foo` can taint

the memory location `p` points to, which may be a location of a return address, a function frame pointer or another pointer.

5.3.1. Analysis of `strcpy()`

A simple but interesting example is `strcpy()`, which copies a NULL-terminated source string to a destination buffer. The string manipulation functions, including `strcpy()`, `strcat()` and `sprintf()`, are known to cause a significant number of buffer overflow vulnerabilities. Our formal reasoning extracts security conditions from the implementation of `strcpy()`. The source code of `strcpy()` and its formal representation are given in Table 14.

Table 14: Source Code and Formal Semantics of `strcpy()`

`char * strcpy (char *dst,`
`  char *src) {`
`  char * res;`
`  res =dst;`
`  while (*src!=0) {`
`1:  *dst=*src;`
`  dst++;`
`  src++;`
`  }`
`2:  *dst=0;`
`  return res;`
`}`
	`mov [res] <- (^ dst) ;`
	`while (~((^ ^ src) is 0)) do`
	`L1:  mov [^ dst] <- (^ ^ src) ;`
	`mov [dst] <- (^dst + 1) ;`
	`mov [src] <- (^ src + 1)`
	`od ;`
	`L2:  mov [^ dst] <- 0 .`

Because the only indirect write operations in the source code are in Line 1 and Line 2, two theorems listed in Table 15 need to be proved. We assume that the NULL-terminator (i.e., the character '`\0`') of the source string src is at the location (`src + srclen`), and that the size of the buffer `dst` is dstsize. Theorem NV1 ensures that before Line L1, the content of the variable `dst` is not tainted; Theorem NV2 ensures that after Line L1, the content of the `srclen`th location in the buffer pointed by dst is not tainted. This condition eliminates the possibility of buffer overflow. NV3 is similar to NV1, but proves the property for the memory state before Line L2 is executed. The security pre-conditions extracted are shown in Table 18.

Table 15: Theorems to Prove for Function `strcpy()`

```
Theorem NV1: If S1 is the store before Line L1, then
          LocT(S1,dst) = false
Theorem NV2: If S2 is the store after Line L1, then
          ExpT(S2, (^dst) + dstsize) = false
Theorem NV3: If S3 is the store before Line L2, then
          LocT(S3,dst) = false
```

Table 16 gives a set of security preconditions extracted in the process of proving the theorems. Among the four preconditions, Condition 4 is well known because of the large number of buffer overflow vulnerabilities caused by string manipulation. This condition has already been documented on Linux MAN page of `strcpy`. Condition 2 indicates the scenario of overlap between `src` and `dst`. This is examined further in Section 5.4.1. Violation of Condition 3 may occur when a program miscalculates the location of a stack buffer, causing the function frame of `strcpy()` to be covered by the buffer and a sample code violating the condition is given in Section 5.4.2.

Table 16: Sufficient Conditions to Ensure the Validity of Theorems NV1 – NV3

1. Initially, the location of `dst` is not tainted.
2. The buffers `src` and `dst` do not overlap in such a way that the buffer dst covers the NULL-terminator of the src string.
3. The buffer dst does not cover the function frame of `strcpy()`, which consists of the locations `&dst`, `&src` and `&res`.
4. `srclen < dstsize`

5.3.2. Analysis of `free()`

We implemented a binary buddy heap management system including function `malloc()` and `free()`. The memory block to be freed is pointed to by pointer `FreedBlock`. The binary buddy heap management algorithm requires the deallocated memory block `FreedBlock` to be merged with its buddy block if the buddy block is also free. The pointer `BuddyBlock` points to the buddy block. `FreedBlock` and `BuddyBlock` are structs of type `HEAP_BLOCK` as shown in Table 17. The `Size` field indicates the size of the memory chunk. The `Busy` field indicates whether the memory chunk is free. Fields `Fwd` and `Bak` are pointers to maintain a doubly-link list of free memory chunks.

Table 17: Indirect Write Statements in `Free()` Source Code

```
typedef struct _HEAP_BLOCK {
    int             Size;           // The size of the block.
    int             Busy;           // Is this block busy?
    struct _HEAP_BLOCK * Fwd,* Bak; // List to the free blocks of
                                    // the same size
} HEAP_BLOCK;
```

There are three lines in the `Free()` function where indirect write operations are performed. Six pointers are involved in the operations, including `FreedBlock`, `BuddyBlock`, `FreedBlock->Fwd`, `FreedBlock->Bak`, `BuddyBlock->Fwd` and `BuddyBlock->Bak`. Table 18 states the theorems to be proved for conditions guaranteeing that pointers are not tainted. The following discussion assumes that the offset of the `Fwd` field in the `HEAP_BLOCK` structure is 2, and that the offset of the `Bak` field is 3. Theorem NV1 ensures that none of the six pointers is tainted before executing any indirect writes.

Table 18: Theorems to Prove for Function `Free()`

Theorem NV1: If S is the store before executing the indirect writes, then	
`(ExpT(S, (^ FreedBlock)) = false)` and	//`FreedBlock` is not tainted
`(ExpT(S, (^ BuddyBlock)) = false)` and	//`BuddyBlock` is not tainted
`(ExpT(S, ^((^ FreedBlock)+2) = false)` and	//`FreedBlock->Fwd` is not tainted
`(ExpT(S, ^((^ FreedBlock)+3) = false)` and	//`FreedBlock->Bak` is not tainted
`(ExpT(S, ^((^ BuddyBlock)+2) = false)` and	//`BuddyBlock->Fwd` is not tainted
`(ExpT(S, ^((^ BuddyBlock)+ 3) = false)`	//`BuddyBlock->Bak` is not tainted

The process of proving the theorems extracted a set of formally specified conditions that guarantee the validity of Theorem NV1. Table 19 describes the conditions. The function `Free()` is safe to be called when the caller function can guarantee these conditions. Violations of condition 1 are unlikely to occur, and the same is true for condition 7. Violations of condition 6 cause the classic double-free errors, and violations of condition 3 and 4 lead to the popular heap buffer overflow vulnerability. An example illustrating violation of condition 2 is presented in Section 5.4.3.

Table 19: Sufficient Conditions to Ensure the Validity of Theorem NV1

1. The memory range of the heap and the memory range of the current function frame do not overlap.
2. Immediately before `Free()` is called, `FreedBlock` points to a location in the range of the heap.
3. Immediately before `Free()` is called, the `Fwd` and `Bak` links of the block of `FreedBlock` are not tainted.
4. All free-chunk double-linked lists are within the heap range, i.e., no `Fwd` or `Bak` links points to any location outside the heap.
5. No `Fwd` or `Bak` pointers in any free-chunk double-linked list are tainted.
6. Immediately before `Free()` is called, `FreedBlock` is not linked in any free-chunk double-linked list.
7. If `BuddyBlock` is freed, then `BuddyBlock` is linked in a free-chunk double-linked list.

5.3.3. Analysis of `printf()`

We implemented a function `Printf()`, similar to the LibC function `printf()`, except that `Printf()`calls its child function `Vfprintf()`, which is a simplified version of LibC function `vfprintf()`.`Vfprintf()`implements the format directives *%%, %d, %s* and *%n*. The total length of `Vfprintf()`is 55 lines. Pointer `p` is used to sweep over the format string format. The argument list is swept over by pointer `ap`.

There are only two lines of indirect write operations in the function (Table 20). Line L1 is to get the last digit of data and save it in the n^{th} position of the buffer `buf`. Line L2 is to assign the character count to the memory location pointed by the current argument. Note that `ap` is the argument list pointer pointing to the current argument.

Corresponding to the two indirect write operations, we need to prove the theorems given in

Table 21. Theorem NV1A ensures that before executing code in Line L1, the memory location hosting the variable n is not tainted. Theorem NV1B ensures that after Line L1, the memory location `buf+10` is not tainted. Theorem NV2 ensures that before Line L2, the expression `(^ ^ ap)` is not tainted, i.e., the memory location pointed by `(^ ap)` is not tainted, i.e., the memory location pointed by the content of variable `ap` is not tainted.

Table 20: Indirect Write Statements in `Vfprintf()` Source Code

```
L1:   buf[n]=data%10+'0';
L2:   *(int*)ap = count ;
```

Table 21: Theorems Need to Prove for `Vfprintf()`

Theorem NV1A: If S is the store before executing Line L1, then
`ExpT(S,(^ n)) = false`
Theorem NV1B: If S is the store after executing Line L1, then
`LocT(S,(buf + 10)) = false`
Theorem NV2: If S is the store before executing Line L2, then
`ExpT(S, (^ ^ ap)) = false`

Theorem NV1A and NV1B are easily proved without the necessity of specifying any preconditions. However, the proof of NV2 could not be established, as the theorem prover was unable to proceed after certain point, which suggested our proof obligations. After proving the obligations as lemmas or specifying them as preconditions, the proof process was able to complete. Eventually, when the theorem is proved, we obtain a list of formally specified preconditions, which are described in Table 22.

Table 22: Sufficient Conditions to Ensure the Validity of Theorem NV2

1. `ap` never points to any location within the current function frame. 2. `*ap` never points to the location of variable ap, i.e., `*ap != &ap`. 3. Suppose the memory segment that ap sweeps over is called *ap_activitiy_range*, no locations within *ap_activitiy_range* are tainted before `Vfprintf()` is called. 4. `*ap` never points to any location within *ap_activitiy_range*.

The four conditions form a set of sufficient conditions, which if satisfied, guarantee that there is no pointer taintedness situation in the analyzed version of `Vfprintf()`. Format string vulnerabilities do not satisfy condition 3 (Table 22). As illustrated in Figure 15, the tainted data (word `0x08049978`) is located in the activity range of `ap`, i.e., `ap` points to this data. For the other three conditions, we are currently unaware of any existing applications violating them. It is the programmers' responsibility to ensure the validity of these hidden assumptions.

5.4. Examples Illustrating Violations of Library Functions' Preconditions

In the previous section, we have given a significant number of preconditions for common library functions. Not all of them are likely to occur in real application code. In this section, we give possible scenarios (constructed examples) in which some of the preconditions detailed in the previous section are violated, and explain how an attacker can exploit them. To the best of our knowledge, these vulnerabilities have not been reported in any real application or described in the literature.

5.4.1. Example of `strcpy()` Violation – Condition 2

Condition 2 in Table 16 for `strcpy()` states that the buffer `dst` does not cover the function frame of `strcpy()`, which consists of *`dst`, `src`* and *`res`*. Otherwise it is possible to overwrite the stack frame of `strcpy()` and modify the address of the *dst* string. Since `strcpy()` can write to the location `(*dst)`, this can be used to write to any memory location, including function pointers, and hence transfer control to malicious code.

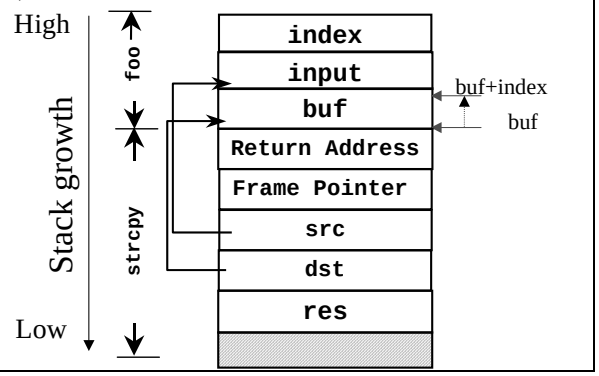
Consider the code sample in Table 23a, in which `buf` and `input` are allocated on the stack in the function frame of `foo()`. The string `input` is obtained from the user and passed as the `src` argument of `strcpy()`. The `dst` argument of `strcpy()` is `buf + index`, where `index` is computed by subtracting the length of *input* from the end of the buffer `buf`. After `strcpy()` is called, the stack frame looks as shown in Table 23b. Assume that the attacker enters an input string longer than 20 bytes as input. Since the input buffer has a size of 100 bytes, this may not cause buffer overflow. However, this makes the value of `index` computed to become negative, which in turn makes `dst` point to a stack location before `buf` and in the function frame of `strcpy()` (thereby violating the pre-condition). In the above example, setting the `index` to (-16) makes `dst` point to the location of itself on the stack. The `strcpy()` code then writes to the location of `(*dst)`, thereby overwriting `dst` itself. Subsequent writes to `(*dst)` then modify the contents of the location pointed to by this new value of `dst`. This allows the attacker to write any value to any memory location, including potentially sensitive locations such as function pointers.

Table 23: Violation of Condition 2 of `strcpy()`

a) Sample Code

```
void foo(  ) {
  int index;
  char input[100];
  char buf[20];
  scanf("%s", input );
  index = 20 –  strlen( input );
  strcpy( buf+index ,input );
}
```

b) Stack Status



The functionality of the code shown in Table 23a is to push data to the end of a buffer. We believe it is possible that applications require such a functionality. For example, a program may need to copy data at the end of a buffer and prefix headers in front the data. The pointer arithmetic shown in Table 23a is an efficient means of implementing such an operation, so we argue that the sample code demonstrates a possible scenario in real applications.

5.4.2. Example of `strcpy()` Violation – Condition 3

In Table 16, condition 3 of `strcpy()` states that `src` and `dst` do not overlap in such a way that `dst` covers the null-terminator of `src`, otherwise the null-terminator of `src` string gets overwritten and the program can go into an infinite loop. This can happen in two ways: by a buffer overflow error or by an inadvertent free error, as illustrated in Table 24a and Table 24b, respectively.

Table 24: Two Cases Depicting Examples of `strcpy()` Condition 3 Violations

a) Buffer Overflow Error

```
char* src = malloc(20);
char* dst = malloc(20);
sprintf(src,"string with > 20
characters");
strcpy(dst, src);
```

b) Inadverent Free Error

```
src = malloc(40);
snprintf(src, 30, "some string of
         30 or more characters");
free( src );
foo = malloc (10);
dst = malloc(20);
strcpy( dst, src );
```

In the first piece of code (Table 24a), two buffers are allocated on the heap and one of them is overflowed. This buffer is then passed as the `src` argument to `strcpy()`,

and the other one as the `dst` argument. Upon running this code multiple times[17], we found that the memory manager consistently allocated nearly consecutive, successive memory addresses to `src` and `dst` respectively. As a result, when the `src` buffer is overflowed, its contents spill into the `dst` buffer, causing it to overlap with the *src* string and cover the null-terminator, leading to a violation of the pre-condition. Note that this can happen even if the source and destination buffers are not nearly contiguous, provided that the address of the destination buffer is greater than the address of the source buffer and the input string is long enough to overflow into the destination buffer.

The `src` and `dst` arguments can also overlap if the destination buffer is allocated from some portion of the source buffer. This situation is illustrated in Table 24b. Here *src* is first allocated on the heap and then freed, which returns the `src` buffer to the free pool. When `malloc()` requests are made subsequently for `foo` and `dst`, the memory manager reuses the block most recently returned to it, namely the `src` buffer, for allocating the buffers `foo` and `dst`. When `strcpy()` is called, `src` and `dst` overlap in such a way that `dst` covers the null terminator of `src`, which is a violation of the pre-condition. In real codes, this can happen as a result of using a buffer that is freed on an infrequently executed path in the code and may not be uncovered during testing.

5.4.3. Example of `free()` Violation – Condition 2

Condition 2 of the `free()` function in Table 19 states that the pointer passed to `free()` must be within the heap range. This arises from the fact that the `free()` function itself does not perform this check. When a block is freed, the `free()` function checks for an integer value at the beginning of the block, which represents the size of the block to be freed. If it finds such an integer, it does the free, irrespective of whether the block is on the heap or not.

Consider what happens when a local buffer on the stack is passed to the `free()` function in Table 25. In this code, the local array `buf` of function `foo()` is passed to the function `print_str()`, which checks if the length of the string passed to it is more than the value `n` specified by the user, and if so, frees the buffer. The pointer `p` which is

[17] We tried it with GNU-LIBC on both x86-linux and Sun Solaris platforms. Our results indicate that this is not an OS or platform specific phenomenon, but a feature of GNU-LIBC.

freed by `print_str()` is aliased to `buf`, which is allocated on the stack in the function frame of `foo()`, leading to a violation of the pre-condition. Since this happens only when the user enters a string of more than 50 characters, it may not be uncovered while testing. In this example, the integer `i`, which is a local variable of `foo()` is present on the stack at the beginning of the block `buf`. The `free()` function assumes that this is the size of the buffer `buf` and attempts to deallocate a block of that size. Since the user also supplies this value, it is possible to free a block of any arbitrary size on the stack, and overwrite the contents of any memory location.

Table 25: Violation of Condition 2 of `free()`

```
void foo() {
   char buf[100];
   int i;
   scanf(“%d”, &i);
   scanf(“%s”, buf);
   print_str( buf  , 50);
}
```

```
void print_str( char* p, int n ) {
   if ( strlen(p) > n ) {
          free( p );
          return;
  }
  printf(stdout, “%s”, p);
}
```

CHAPTER 6

DEFEATING SECURITY ATTACKS BY POINTER TAINTEDNESS DETECTION

6.1. Architectural Support for Pointer Taintedness Detection

The theorem-proving-based static analysis technique presented in CHAPTER 5 helps in formal reasoning about pointer taintedness, and thus exposes potential security vulnerabilities. This chapter shows that the concept of pointer taintedness also enables an effective runtime detection technique, an architecture-level mechanism completely transparent to applications. Unlike the static analysis technique, the runtime detection technique cannot expose potential security vulnerabilities; it can only detect attacks when they occur. In this sense, the static analysis technique discussed earlier and the runtime detection technique in this chapter are not competing, butcomplementary.

The runtime detection mechanism is implemented as multiple components. In order to implement the taintedness-aware memory model presented in CHAPTER 5, the existing memory system is extended by adding an additional taintedness bit to each byte. The taintedness bit is set whenever data from input devices is copied into the memory. Within the processor execution engine, the taintedness bit is propagated when tainted data are used for an operation. Whenever a tainted word is used as an address value for memory access (data or code accesses), an exception is raised by the processor. The operating system then handles the exception and stops the current process to defeat the ongoing attack.

6.1.1. Extended Memory Architecture

The memory system architecture is extended to support the notion of taintedness. A taintedness bit is associated with each byte in memory. When a memory word is accessed by the processor, the taintedness bits are passed through the memory hierarchy together with the actual memory words. L2 and L1 caches and data storage within the processor (registers and buffers) are also extended with the additional taintedness bits.

The detection mechanism is designed on top of the extended memory model. Although the underlying principle is general enough to be applicable to other architectures, the discussion is given in the context of *SimpleScalar* RISC architecture. Figure 17 gives the enhancements of the pointer taintedness detection mechanism implemented as extensions of *SimpleScalar*.

6.1.2. Taintedness Tracking

When a program performs operations using its data from memory, the taintedness bit should be propagated. The processor pipeline is modified to track taintedness. In general, any CPU operation that uses tainted data as source should produce a tainted result. This mechanism is similar to the ones proposed in [26] and [73].

We distinguish between memory operations and ALU operations. A memory load operation moves data from memory to a processor register, and a store operation moves data from a processor register to memory. Corresponding to the one-bit extension to each memory byte, the processor registers are also extended to include one taintedness bit for each byte. For each load instruction, the data bits as well as the taintedness bits are copied from memory to register along the load path. Similarly, store instructions write normal data bytes as well as taintedness bits to the memory along the store path.

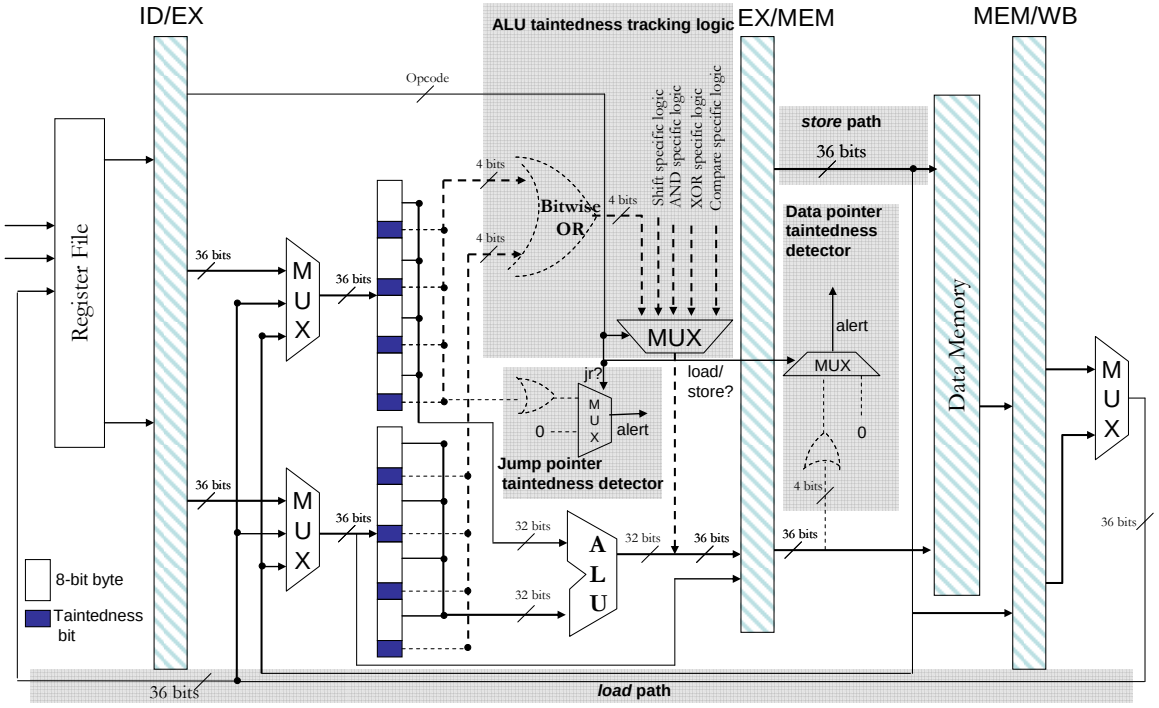


Figure 17: Architecture Design of Pointer Taintedness Tracking and Detection

ALU instructions are responsible for propagating taintedness between registers. The propagation is implemented by the ALU taintedness tracking logic (shown as a shaded area in Figure 17). With the few exceptions noted below, the ALU taintedness works as follows: for operations with two source operands, the taintedness bits of a resultant register are obtained by the bitwise OR of the corresponding taintedness bits in the source operand registers. For example, after executing *ADD R1,R2,R3,* R1 is tainted if and only if R2 is tainted or R3 is tainted.

The following exceptional cases require special handling. (1) Shift instructions cause taintedness to propagate within the operand register. If a byte in the operand register is tainted, then the taintedness bit of its adjacent byte along the direction of shifting is set to 1. (2) The taintedness bits of any byte AND-ed with an untainted zero are cleared, because the resulting byte value is constant 0, regardless of user input. (3) The compiler idiom *XOR R1,R2,R2* is frequently used to assign constant 0 to the target register R1. The taintedness bits in R1 are cleared as a result. This idea is borrowed from previous techniques [26] and [73]. (4) Compare instructions are used for data range checking. If a tainted register R1 is compared with some untainted data in R2, the taintedness bits in R1 are cleared after the operation. The rationale is that programmers often write input validation code to check certain safety properties. The validation code is in the form of compare instructions. For application compatibility, any data that undergoes validation is trusted after such an operation. This could potentially lead to missed detection (a.k.a. false negatives),or example, in situations in which the program does check user input values but the check algorithm is flawed. The false negative scenarios are discussed in Section 6.2.3.

Table 26 summarizes the taintedness tracking logic. The ALU taintedness tracking logic consists of a multiplexer (MUX) selecting from four sources of input based on the opcode of the current instruction. These multiplexer inputs correspond to the five types of ALU instructions listed above.

Table 26: Taintedness Propagation by ALU Instructions

ALU Instruction Type	Taintedness Propagation
ALU instructions except *shift*, *compare*, and *AND*, e.g., *op R1,R2,R3*	Tainedness of R1 = (Taintedness of R2) or (Taintedness of R3).
Shift instruction	If a byte in the operand is tainted, the taintedness bit of its adjacent byte along the direction of shifting is set to 1.
AND instruction	Untaint each byte AND-ed with an untainted zero.
XOR R1,R2,R2	Taintedness of R1 = 0000.
Compare instruction	Untaint every byte in the operands of the compare instruction.

6.1.3. Attack Detection

In general, whenever a tainted data value is used for memory access, an alert should be raised. The proposed detection mechanism is described using the instruction set architecture of the *SimpleScalar* processor simulator. In *SimpleScalar*, only the load/store instructions and the jump instruction *JR* (i.e., jump to the address in a register) can dereference a pointer, which is stored in a register. The jump pointer taintedness detector is placed after the *ID/EX* (instruction decode/execution) stage when the jump target register value is available. The four taintedness bits in the target register are OR-ed. If any byte in the word is tainted, the output of the OR-gate is 1 and the instruction is marked as malicious. The detector of tainted pointers for load/store instructions is placed after the *EX/MEM* (execute/memory) stage; here the four taintedness bits of the address word are inputted into an OR-gate, and the instruction is marked as malicious if the output of the gate is 1 and the instruction opcode is load or store. The actual security exception is raised in the pipeline's retirement stage. Retirement of an instruction marked as malicious causes the pipeline to raise a security exception. The operating system can then terminate the process and stop the ongoing intrusion.

6.1.4. Taintedness Initialization

Any data received from an external device that can potentially be controlled by a malicious user are considered tainted, e.g., input coming from network, file system, keyboard, command line arguments, and environmental variables. All such data are passed from an external source to the program buffer through operating system calls. The system call implementations are modified to mark every byte in the buffer as tainted when it is returning from kernel space to user space. This can be implemented by adding every word in the buffer to a special register *RT*. The value of *RT* is always 0, but every

taintedness bit of *RT* is 1. In the current implementation, the system call module of the *SimpleScalar* simulator is modified for this purpose. In particular, all data delivered to the application through the SYS_READ (local I/O) and SYS_RECV (network I/O) are marked as tainted. These two system calls are invoked by most input functions in C library, such as `scanf(), fread(), recv()` and `recvfrom()`.

In summary, three subsystems in the *SimpleScalar* simulator are modified to implement the algorithm: (1) The memory subsystem is extended with the taintedness bits in the memory, the cache, and the register file. (2) The original system call implementation is modified so that `SYS_READ` and `SYS_RECV` mark every byte in the receiving buffer as tainted. (3) The instruction pipeline is extended to implement taintedness calculation, propagation, and detection.

6.2. Evaluation

The proposed architecture has the following properties: (1) high coverage in detecting attacks tampering with both control and non-control data; (2) transparency to applications, i.e., the detection does not rely on any internal knowledge of the applications, e.g., buffer sizes, variable upper bounds, or program semantics; (3) no known false positives; and (4) very small space overhead and performance overhead. These properties are evaluated by running synthetic programs, real network applications, and SPEC benchmarks on the proposed architecture.

6.2.1. Security Protection Coverage

The pointer taintedness detection technique provides a significant improvement in security coverage by protecting applications from both control data attacks and non-control-data attacks. The security coverage of existing control-flow integrity based protections was evaluated against control data attacks only. This section shows that non-control-data attacks do exist and can cause the same level of security compromise in many real applications. For a fair comparison, several applications are employed that were previously used to assess the existing techniques.

6.2.1.1. Synthetic Vulnerable Programs

The effectiveness of the proposed approach is first demonstrated on a number of synthetic functions that are vulnerable to stack buffer overflow, heap corruption, and format string attacks respectively (Table 27).

Table 27: Synthetic Vulnerable Programs

Stack Buffer Overflow	**Heap Corruption Attack**	**Format String Attack**

`void exp1() {`
`  char buf[10];`
`  scanf("%s",buf);`
`}`
	`void exp2() {`
	`  char * buf;`
	`  buf = malloc(8);`
	`  scanf("%s",buffer);`
	`  free(p)`
	`}`
		`void  exp3(int s) {`
		`  char buf[100];`
		`  recv(s,buf,100,0);`
		`  printf(buf);`
		`}`

Detection of stack buffer overflow. When a string of "a" characters of 24 bytes is passed to `exp1()` running on this architecture, an alert is raised at the return instruction (i.e., `JR $31` on *SimpleScalar*) of `exp1()`, which indicates that the return address is tainted as `0x61616161`, corresponding to four "a" characters in the input.

Detection of heap corruption. Function `exp2()` contains a heap overflow vulnerability. An attack is launched by inputting 12 "a" characters to the 8-byte buffer. When the buffer is freed, a *load-word* instruction `LW $3,0($3)`, which is in function `free()`, raises an alert. As described in Section 3, a statement executed in `free()` is `B->fd->bk=B->bk`. When the alert is generated, register `$3` equals `B->fd`, which is a tainted word *0x61616161* due to the buffer overflow condition. Because the detected instruction attempts to dereference register `$3` (i.e., the `0($3)` indirect addressing mode) when its value is tainted, the alert is raised.

Detection of format string attack. The effectiveness of detecting format string attacks is demonstrated by function `exp3()`. The function receives the string `abcd%x%x%x%n` from the socket. When `printf()` is called, a *store-word* instruction `SW $21,0($3)` in `vfprintf()` raises an alert. This store instruction is compiled from the statement `*ap=count` described in Section 5.1.1, where the value of `ap` is in register `$3` and the value of count is in register `$21`. When the alert is raised, the value of register `$3` in `0($3)` dereference is `0x64636261`, corresponding to the first four bytes of the input string, "`abcd`".

6.2.1.2. Real-World Network Applications

The three examples discussed in the previous section demonstrate that pointer taintedness detection can defeat many types of memory corruption attacks. This section presents results from testing real-world attacks against network applications running on the *SimpleScalar* augmented with pointer taintedness detection capability. In addition, the *SimpleScalar* processor simulator is extended to support network socket applications. The enhancement allows us to run many real-world network server applications. Both control data attacks and non-control-data attacks are used for the evaluations. The pointer taintedness detection technique succeeds in defeating both types of attacks.

WU-FTPD format string attack. Similar to the WU-FTPD attack described in CHAPTER 4, an attack is constructed to exploit the format string vulnerability to overwrite an integer word representing the ID of the login user. With a proper protection, this attack would escalate the attacker's privilege to the root privilege, offering the attacker a full control on the file `/etc/passwd` so that he/she can upload a different version of this file. After writing a malicious entry such as "`alice:x:0:0::/home/root:/bin/bash`" in the new version, the attacker leaves a backdoor to login later as Alice, who possesses root privileges.

Now WU-FTPD is run on the proposed architecture. Table 28 shows the attack/detection steps. When the FTP server is ready to accept user input, the attacker (the FTP client) first authenticates to the server using *USER* and *PASS* commands, then issues a `SITE EXEC` command to exploit the vulnerability. The target integer word representing the user ID is located in the address `0x1002bc20`, so the command used to overwrite this word is:

```
site exec \x20\xbc\x02\x10%x%x%x%x%x%n
```

Immediately after the attack sends the malicious `SITE EXEC` command, the pointer taintedness detector raises an alert indicating that the instruction `SW $21,0($3)` dereferences a tainted value in register `$3`. The value of the register is `0x1002bc20`, the same as the one specified by the attacker as the target address to overwrite. The FTP server is stopped when the alert is raised, which effectively prevents the attack from succeeding.

Table 28: Attacking WU-FTPD on the Proposed Architecture

FTP Server	`220 FTP server (Version wu-2.6.0(60) Mon Nov 29 10:37:55 CST 2004) ready.`
FTP Client	`user user1`
FTP Server	`331 Password required for user1 .`
FTP Client	`pass xxxxxxx (the correct password of user1)`
FTP Client	`site exec \x20\xbc\x02\x10%x%x%x%x%x%n`
Alert	**`44d7b0: sw $21,0($3) $3=0x1002bc20`**

NULL HTTPD heap corruption attack. The attack discussed in CHAPTER 4 is used to test the detection effectiveness of the proposed architecture. The attack is an attempt to exploit the heap corruption vulnerability to overwrite CGI-BIN configuration string as `/bin`, allowing `/bin/sh` to be executed as a CGI program with the root privilege.

To test the effectivenss of detection, *NULL HTTPD* is run on the proposed architecture. The attacker sends a POST command with a negative *Content-Length* value (-800) to cause a heap overflow condition on the server. If the attacker attempts to overwrite the CGI-BIN configuration, which is located in the address 0x1001717d, four characters `\x7d\x71\x01\x10` need to be embedded in the POST command to precisely overwrite the heap free chunk doubly-linked list. If the overflowed buffer were freed, the corruption of CGI-BIN configuration would have been accomplished. Due to presence of the pointer taintedness detector, during the execution of function `free()`, the instruction `LW $3,0($3)` raises an alert to stop the application process. The value of register `$3` is `0x1001717d`, the address of the CGI-BIN configuration string.

Table 29: Attacking NULL-HTTPD on the Proposed Architecture

HTTP Server	`HTTP Daemon Ready`

HTTP Client	`POST / HTTP/1.0`
	`Content-Length: -800`
	`AAAAAAAAAAAAAAA…AAAAA\x7d\x71\x01\x10….`

Alert	**`409650: LW $3,0($3), $3=0x1001717d`**

GHTTPD stack overflow attack. An attack is constructed to exploit the stack buffer overflow bug in GHTTPD. It is very similar to the attack presented in CHAPTER 4: When GHTTPD runs on the proposed architecture, a malicious request `GET AAAAAA…AAAAAA\x94\x3e\xff\x7f↵↵/cgi-bin/../../../../bin/sh` is sent to the server. The first part of the request `AAAAAA…AAAAAA \x94\x3e\xff\x7f` is parsed as a URL. However, due to the buffer overflow vulnerability, the last four bytes overwrite the pointer of the URL to `0x7fff3e94`,

which is the address of the second part of the string, `/cgi-bin/../../../../bin/sh`. Without the protection provided by our architecture, this would force the server to run `/bin/sh` with root privileges. The pointer taintedness protection algorithm effectively stops the attack when the tainted URL pointer is dereferenced in an `LB` instruction. The alert indicates that register $4 has a tainted value *0x7fff3e94* as specified in the malicious request, which is about to be dereferenced.

Table 30: Attacking GHTTPD on the Proposed Architecture

HTTP Server	`HTTP Daemon Ready`
HTTP Client	`GET AAAA…AAAA\x94\x3e\xff\x7f↵↵/cgi-bin/../../../../bin/sh`
Alert	**`405e58: LB $7,0($4),  $4=0x7fff3e94`**

Traceroute double free attack. Certain versions of LBNL *traceroute* are vulnerable to an attack involving freeing of a heap buffer not allocated by `malloc()` [44]. When *traceroute* is executed with the arguments "`-g x -g y`", `savestr()` is called twice to parse arguments "`-g x`" and "`-g y`". `savestr()` reduces calls to *malloc()* by preallocating heap space and performs self buffer management when it is invoked subsequently. After "`-g x`" is parsed and `savestr()` is called, the pointer to the block used by `savestr()` is released using `free()`. When "`–g y`" is interpreted, `savestr()` is called again, and the result is written to the block of already freed memory. Like for "`-g x`", `free()`is called, but this time on a region that has already been released in the first `free()` call. *Traceroute* crashes because `free()` is using an invalid pointer in an invalid `malloc()` header. A malicious user can take over *traceroute* using the double free attack method: it corrupts pointers used by `malloc/free`, then forces *traceroute* to overwrite critical program data or execute malicious code.

This experiment uses the command line `traceroute –g 123 –g 5.6.7.8`. Without the detection mechanism, this results in a successful takeover. With the detection mechanism, an alert is generated at a store-word instruction inside *free()* because `0x333231` is a tainted value when it is dereferenced as a pointer.

6.2.2. Evaluation of False Positives

Along with system security, a crucial criterion of defensive systems is the false positive rate, i.e., the likelihood that the system raises an alert when there is no attack.

The network applications discussed in Section 5.1 run smoothly on the proposed architecture without generating any alert when there is no attack. In order to more thoroughly evaluate the false positive rate of the architecture on real applications, six integer applications from SPEC 2000 are run, of which only their binary executables are available. These applications are BZIP2, GCC, GZIP, MCF, PARSER and VPR, and the default test cases are provided by SPEC 2000. Since none of the test cases is a malicious attack, no alert should be generated during the execution of these programs. Table 31 shows the results of this test: the total size of these programs is 6586KB, the total number of input bytes during the execution of the benchmarks is 2186KB, and the total number of instructions executed is 15,139 million. During the execution of these programs, not a single alert is raised. This experiment is a good indicator that one can expect very few (or even no) false positive when the proposed technique is deployed in real systems.

Table 31: Test False Positive Rate Using SPEC 2000 Benchmark Programs

	BZIP2	GCC	GZIP	MCF	PARSER	VPR	**Total**
Program size	321KB	4184KB	485KB	304KB	595KB	697KB	**6586KB**
Total number of input bytes	1048KB	77.7K	282KB	39.2KB	743.0KB	6.4KB	**2186KB**
Total number of instructions	5,951M	110M	6,926M	1,653M	389M	108M	**15,139M**
Alert generated?	No	No	No	No	No	No	**No**

6.2.3. False Negative Scenarios

False negative scenarios of a defensive technique are the situations in which an attack escapes detection. Although pointer taintedness architecture detects a larger set of memory corruption attacks than existing control-flow integrity based protections, it does not provide 100% security coverage. This section shows some synthetic cases where certain degree of damage can be done to the system running on top of the proposed architecture.

Integer overflow attacks resulting in an out-of-boundary array index. Integer overflow is often due to a programmer's misinterpretations of signed, unsigned, long and

short integers. When a programmer converts integers between these types, the resulting values can be inconsistent with the programmer's expectations. Table 32(A) shows a vulnerable function where an unsigned integer *ui* is assigned to a signed integer *i*. Lines 2 and 3 perform an array index boundary check to ensure that `i` does not exceed the array size. This comparison statement untaints `i` because it has been boundary checked. However, an attacker can input a very large unsigned integer `ui` to the function. When `ui` is assigned to the signed integer `i`, `i` becomes negative. Line 4 uses `i` as the array index, allowing the attacker to overwrite any memory address lower than the address of array. Neither this technique nor the existing control data protection techniques stop integer overflow attacks from corrupting memory. The integer overflow vulnerability differs from other memory corruption vulnerabilities because the integer value is intended to be the array index, while in other vulnerabilities, the values being dereferenced are not supposed to be pointer values: they can be embedded in FTP command, HTTP request, and IP address. To defeat the attack, the bound check must be implemented correctly. Unfortunately, it is very difficult, if not impossible, on the hardware level, to transparently perform the check without knowledge of application semantics.

Buffer overflow attacks corrupting critical flags. Table 32(B) depicts user authentication functionality, where a flag `auth` is defined to indicate whether a user is authenticated. After Line 3 sets this flag by calling `do_auth()`, the buffer overflow vulnerability in Line 4 can be exploited to overwrite the authenticated flag to 1. Line 5 grants access to the user according to the `auth` flag, and therefore an attacker can get the access without successful authentication. The attack cannot be detected by this technique, as the attack simply overflows a buffer to corrupt an integer following it, and no pointer is tainted during the attack.

Table 32: False Negative Scenarios

(A) Integer overflow causing array index out of boundary	(B) Buffer overflow causing critical flags to be corrupted	(C)Format string attack causing information leak

`void foo(`
`  unsigned int ui) {`
`1:  int i = ui;`
`2:  if (i >= ArraySize)`
`3:  i = ArraySize – 1;`
`4:  array[i] = 1;`
`}`
	`void bar () {`
	`1:  int auth;`
	`2:  char buf[100];`
	`3:  auth =  do_auth();`
	`4:  scanf("%s",buf);`
	`5:  if (auth)`
	`  grant_access();`
	`}`
		`void leak() {`
		`1:  int secret_key;`
		`2:  char buf[12];`
		`3:  recv(s,buf,12,0);`
		`4:  printf(buf);`
		`}`

Format string attacks causing information leaks. Although this technique prevents the attacker from overwriting data through a format string attack, Table 32(C) shows that such a vulnerability could allow the attacker to get private information from memory data regions such as the stack. Function `leak()` defines an integer `secret_key` on the stack. A user input buffer `buf` is passed to `printf()` as the format argument. It has been shown that if the attacker sends `abcd%x%x%x%n` to the buffer, an alert is raised because the `%n` directive attempts to dereference a tainted pointer. However, if the input is `%x%x%x%x`, the attacker can read the top four words on the stack, including the `secret_key`. Such an information leak attack can be used for future security compromises not based on memory corruptions, for example, attacks to steal user passwords and secret random seeds.

Despite these false negative scenarios, the proposed technique substantially improves security coverage because (1) most attacks corrupting both control data and non-control data can be effectively defeated, (2) the false negative scenarios are in general not defeatable by any generic runtime detection technique that we are aware of, and (3) the false negative scenarios are rare in the real world.

Effectively exploiting buffer overflow vulnerabilities without corrupting any pointer is also challenging for attackers, because only a limited number of words following the buffer can be overwritten. For stack overflow, the critical flag must be in the same frame as the buffer being overrun. For heap overflow, this limit is guarded by the locations of the free-chunk links following the buffer. Once the overflowed data exceeds the limit, this technique raises an alert because the return address or the links are tainted. The technique cannot prevent information leak damage in format string attacks, but their severity is expected to be much lower than for memory corruptions.

One direction that can potentially reduce the false negative rate is to sacrifice the transparency of the proposed taintedness detection architecture. Programmers can be asked to annotate important data structures that should never be tainted. The annotated data can then be monitored by the architecture. Then, whenever an annotated structure becomes tainted, an alert is raised.

6.2.4. Architectural Overhead

Area overhead. The proposed method will incur some area overhead in a microprocessor and in the overall memory system. Within the processor, the data path between pipeline stages needs to be expanded to accommodate the taintedness bit for each byte of data. The internal physical registers, buffers, and other data structures should be expanded, as should the data bus between the processor, caches, and physical memory banks. Physical memory banks should also increase in width to accommodate the taintedness bit.

Performance overhead. The proposed detection mechanism should not cause slowdown or longer cycle time in the pipeline of a modern processor. This is because the propagation of the taintedness bits through load, store, and ALU operations are not on the critical path of these operations. For example, in executing add r1, r2, r3, the taintedness tracking algorithm need only perform a logic OR operation, which can be carried out in parallel with the add operation. In fact, the logic OR operation takes less time than the add operation to complete, so the taintedness tracking algorithm will not increase clock cycle time for the ALU pipeline stage. For load and store operations, the taintedness bit is directly copied from source to destination and therefore can be performed at wire speed. At the retirement stage, the processor checks whether a memory access (load/store or control flow transfer instructions) uses tainted address values, which is a single bit operation. Again, the checking is simpler than the normal operations required for instruction retirement. Based this analysis, we believe that the operations for the pointer taintedness algorithm do not add pipeline stages or increase cycle time.

Software processing overhead. The operating system kernel requires changes. In particular, the kernel should mark data originating from input system calls as tainted. This can be done before the operating system passes such data back to user space. If we

assume that tainting a byte requires an additional instruction, the percentage of additional instructions executed by a SPEC benchmark program is between 0.002% and 0.2% based on the data in Table 31. Since the current prototype is based on a processor simulator, the discussed operating system enhancement is implemented via system call interception. Actual modification of the operating system requires further investigation.

CHAPTER 7

COMBINING STATIC ANALYSIS AND RUNTIME DETECTION

7.1. Taking Advantage of Static Analysis and Runtime Detection

The theorem-proving-based static analysis technique and the runtime detection technique have both advantages and disadvantages. The static analysis technique does not require any physical modification to the processor architecture, but it requires a substantial amount of human interventions to perform the theorem-proving task. The runtime detection technique is fully automatic, but the modification to the processor, especially the taintedness bit extension to the memory system, still imposes a deployment difficulty in the near future. This chapter demonstrates a technique combining the static analysis and the runtime detection technique. This technique provides a higher degree of automation in deriving security specifications, and it relies on runtime checking to enforce these specifications. Because the reasoning of pointer taintedness is performed statically, there is no need to physically implement the taintedness-aware memory system, which can offer an easier deployment on the current architecture.

7.2. Deriving Security Specifications for Functions

7.2.1. Flowchart Depicting the Technique

To provide a higher degree of automation, a *verification condition generation* technique (a.k.a., *VC generation*) is used, which is essentially a backward reasoning process: for each instruction containing a pointer dereference, an initial VC is specified stating that "the value of this pointer at this instruction should not be tainted." Assuming the program counter `pc` is `n` at this instruction, this VC is denoted as VC(n). The functionality of the VC generator is to start with VC(n) to derive VC(n-1), VC(n-2) …. VC(0), based on the semantics of instructions in the analyzed function. VC(0) is the security specification that needs to be satisfied at the entry of the function in order to

ensure VC(n) at instruction n. In other words, since VC(n) states that instruction n does not dereference a tainted value, VC(0) is the sufficient condition that eliminates the pointer taintedness possibility at instruction n.

Figure 18 gives a flowchart depicting the process of extracting security specifications. First, C source code of the function to be analyzed is compiled to its formal semantic representation, which is a simple assembly-like language defined as a Maude module. This language is referred to as language L. Based on the program in language L, for each pointer dereference in an assignment instruction, the VC generator automatically specifies an initial VC stating that the pointer is not tainted. Finally, the VC generator gets into an iteration to generate VCs for each instruction.

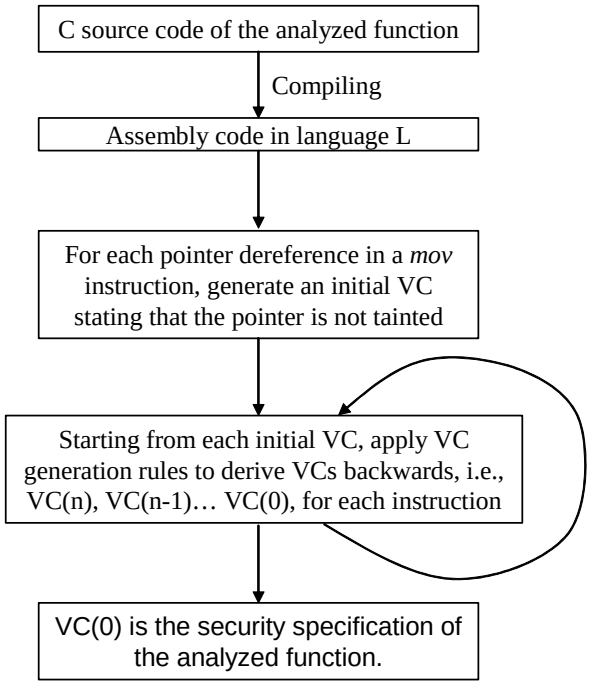


Figure 18: Flowchart of Security Specification Extraction

7.2.2. Language L

Language L is an assembly-like language, consisting of a few program constructs – move instruction, branch instruction, integer, Boolean, dereference operation and ALU operations. The difference between Language L and the formal semantic definition discussed in CHAPTER 5 is that Language L consists of branch instruction, but no *while* and *if* instructions. Language L is more suitable for performing VC generation instruction by instruction. The semantics of Language L instructions are given in Table 33.

Table 33: Language L Instructions

Instruction	Semantics (i.e., VC generation rule)
`inst(n): mov [E1] <- E2`	Assign the evaluation result of `E2` to the memory location addressed by the evaluation result of `E1`. The semantic definition similar to the one given in Section 5.2
`inst(n): branch B m`	If Boolean expression `B` is evaluation to true, jump to instruction `m`; otherwise, continue on instruction `n+1`. `VC(n)=(~ B -> VC(n+1)) && (B -> VC(m))`
`inst(n): no-op`	Null operation `VC(n)= VC(n+1)`

7.2.3. Compiling C Program to L Program

A compiler was implemented to automatically compile a function written in C to the representation of Language L. Table 34 gives the source code of function `strcpy()` and the object code automatically generated by the compiler. The compilation is straightforward. For example, C assignment `res = dst` corresponds to instruction `inst(1)`, where `res` and `dst` are the memory addresses of variables `res` and `dst` in C program. The while statement "`while (*src!=0)`" is translated to `inst(2), inst(3), inst(7)` and `inst(8)`. Semantically, `inst(3)` means that if double-dereferencing the address of variable `src` (i.e., `^ ^ src`) yields 0, jump to `inst(8)`. The symbol `~` is the "`not`" operator. C statement "`return res`" is translated to `inst(10)` and `inst(11)`, where `strcpy` represents the return value of the function `strcpy()`.

Table 34: Source Code of `strcpy()` and Object Code in Language L

```
char * strcpy (                  inst(1) = mov  [res] <- ^ dst .
  char *dst, char *src) {        inst(2) = no-op .
    char * res;                  inst(3) =     branch (~(~(^^src is 0))) 8 .
    res =dst;                    inst(4) =     mov  [^ dst] <- ^ ^ src .
    while (*src!=0) {            inst(5) =     mov  [dst] <- (^ dst) + 1 .
      *dst=*src;                 inst(6) =     mov  [src] <- (^ src) + 1 .
      dst++;                     inst(7) =     branch true 2 .
      src++;                     inst(8) = no-op.
     }                           inst(9) =  mov  [^ dst] <- 0 .
    *dst=0;                      inst(10) = mov  [strcpy] <- ^ res .
    return res;                  inst(11) = branch true 12 .
}                                inst(12) = no-op .
```

7.2.4. VC Generation

To eliminate pointer taintedness, it must be ensured that at every indirect write instruction, the target address does not contain a tainted value. For example, in function `strcpy()`,at `inst(4)`, `^ dst` must not be a tainted value; and at `inst(9)`, `^ dst` must not be a tainted value. These are formally specified as "`VC(4) = TAINTEDNESS(^ dst) is false`" and "`VC(9) = TAINTEDNESS(^ dst) is false`", where `TAINTEDNESS(x)` is a function to test the taintedness of expression `x`. The VC generator then starts the backward reasoning, which automatically extracts the loop invariant. In this case, the loop invariant is trivially "`VC(3) = TAINTEDNESS(^ dst) is false`". The next section presents two non-trivial examples to show the VC generation process.

7.3. Case Study

7.3.1. Function `Vfprintf()`

Static Extraction of Security Specification. Table 35 gives our implementation of `Vfprintf()`, which is a simplified version of LibC standard function `vfprintf()`. `Vfprintf()` implements the functionalities of `%%, %s, %d, %n` format directives. The function consists of 57 lines of C statements. The compiler automatically compiles `Vfprintf()` to the object code shown in Table 36. Only two indirect write instructions are present in the code. For `inst(31)`, the VC generator specifies "`VC(31) = TAINTEDNESS(buf + ^n) is false`" for `inst(31)` and "`VC(56) = TAINTEDNESS(^p) is false`" for `inst(56)`. The VC generation process starting from `VC(31)` ends with "`VC(27) = true`", indicating that `VC(31)` always holds no matter how `Vfprintf()` is called. This is because buf is an address constant generated by the compiler, which is not a tainted value, and because n is initialized to be 0 and incremented by 1, not a tainted value either. Therefore, the VC generation process naturally eliminates the possibility of pointer taintedness in `inst(31)`. However, `VC(56)` is not a trivial condition. The VC generator automatically extracts "`VC(8) = (~ (^ state is 1) && ^ ^ p is 110) -> TAINTEDNESS(^ ap) is false`". This suggests that at the entry of the while loop body, if state is not 1

(indicating a '%' character being the most recent character in the format string) and `*p` is 'n' character (110 is the ASCII code of 'n'), then pointer ap should not be a tainted value. This is because `Vfprintf()` will encounter "%n" and use `*ap` as a pointer value in this iteration. Note that deriving this precise loop invariant `VC(8)` is a fully automatic process.

Table 35: C Source Code of `Vfprintf()`

```
int Vfprintf (char *s, char *format,
char * ap)
{
  char * p, *q;
  int done,state,data,n;
  char buf[10];
  p=format;
  done=0;
  if (p==0) return 0;
  state=1;
  while (*p != 0) {
    if (state==1) {
          if (*p=='%') state=0;
          else
                  done++;
    }
    else {
        if (*p=='%') {
                done++;
        }
        else if (*p=='d') {
            data=*ap;
            if (data<0) {
                    done++;
                    data=-data;
            }
            n=0;
            while(data>0 && n< 10 ){
              *(&buf+n)=data%10+'0';
              data/=10;
              n++;
            }
            while (n>0) {
                    n--;
                    done++;
            }
        }
        else if (*p=='s') {
                q=*ap;
                if (q==0) break;
                while (*q!=0) {
                        done++;
                        q++;
                }
        }
        else if (*p=='n') {
                q = *ap;
                *(int *)q = done;
                done++;
        }
        else { done++;
        }
        state=1;
    }
    p++;
  }
   return done;
}
```

Table 36: Object Code of `Vfprintf()` in Language L

```
inst(1) = mov  [p] <- ^ format
inst(2) = mov  [done] <- 0
inst(3) =  branch (~(^ p is 0)) 6
inst(4) =  mov  [Vfprintf] <- 0
inst(5) =  branch true #return
inst(6) = no-op
inst(7) = mov  [state] <- 1
inst(8) = no-op
inst(9) =   branch (~(~(^ ^ p is 0))) 66
inst(10) =    branch (~(^ state is 1)) 18
inst(11) =     branch (~(^ ^ p is 37)) 14
inst(12) =     mov  [state] <- 0
inst(13) =     branch true 16
inst(14) =     no-op
inst(15) =     mov  [done] <- (^ done) + 1
inst(16) =     no-op
inst(17) =     branch true 63
inst(18) =     no-op
inst(19) =     branch (~(^ ^ p is 37)) 22
inst(20) =      mov  [done] <- (^ done) + 1
inst(21) =      branch true 61
inst(22) =     no-op
inst(23) =     branch (~(^ ^ p is 100)) 42
inst(24) =      branch (~(^ data < 0)) 27
inst(25) =      mov  [done] <- (^ done) + 1
inst(26) =      mov  [data] <- - ^ data
inst(27) =      no-op
inst(28) =       mov  [n] <- 0
inst(29) =       no-op
inst(30) =       branch (~(0 < ^ data
                   && ^ n < 10)) 35
inst(31) =      mov  [( buf + ^ n)] <- (
                   (^ data rem 10) + 48)
inst(32) =      mov  [data] <- ^ data /10
inst(33) =      mov  [n] <- (^ n) + 1
inst(34) =      branch true 29
inst(35) =      no-op
inst(36) =      branch (~(0 < ^ n)) 40
inst(37) =       mov  [n] <- (^ n) - 1
inst(38) =       mov  [done] <- (^ done) + 1
inst(39) =       branch true 35
inst(40) =      no-op
inst(41) =      branch true 61
inst(42) =     no-op
inst(43) =     branch (~(^ ^ p is 115)) 53
inst(44) =       mov  [q] <- ^ ^ ap
inst(45) =       branch (^ q is 0) 40
inst(46) =       no-op
inst(47) =       branch (~(~(^ ^q is 0))) 51
inst(48) =        mov  [done] <- (^done)+ 1
inst(49) =        mov  [q] <- (^ q) + 1
inst(50) =        branch true 46
inst(51) =        no-op
inst(52) =        branch true 61
inst(53) =      no-op
inst(54) =      branch (~(^ ^ p is 110)) 59
inst(55) =       mov  [q] <- ^ ^ ap
inst(56) =       mov  [^ q] <- ^ done
inst(57) =       mov  [done] <- (^ done) + 1
inst(58) =       branch true 61
inst(59) =       no-op
inst(60) =       mov  [done] <- (^ done) + 1
inst(61) =      no-op
inst(62) =      mov  [state] <- 1
inst(63) =  no-op
inst(64) =     mov  [p] <- (^ p) + 1
inst(65) =     branch true 8
inst(66) = no-op
inst(67) = mov  [Vfprintf] <- ^ done
inst(68) = branch true 69
inst(69) = no-op
```

Runtime Checking the Security Specification. Having the formally extracted security specification `VC(8)`, it is straightforward to specify a runtime assertion to enforce it (Table 37) Because format string vulnerability is due to pointer taintedness, deploying this assertion can defeat format string attacks.

Table 37: `Vfprintf()` with a Runtime Assertion

```
int Vfpintf(FILE *s, const char *format, va_list ap) {
   …
   while (*p != 0) {
      assert (!(*p==110 && state != 1 && !UNTAINTED(ap)));
      //statically extracted specification
       …
   }
}
int Printf (const char *format,..)
{   …
     return Vfprintf (stdout, format, arg);
}
```

```
void main() {
 Printf("string=%s\ni=%d\n%n",buf,i,&j);
 scanf("%s",buf);
 Printf(buf);
}
```

```
unsigned int stack_top;
mov %esp, stack_top;
ADD_UNTAINTED_ADDR (stack_top-4);
```

```
REMOVE_UNTAINTED_ADDR (stack_top-4);
```

The assertion requires three basic procedures, which can be implemented by software or dedicated hardware. `ADD_UNTAINTED_ADDR(p)` adds the address `p` to a set of "untainted" addresses, indicating that the content in `p` is not tainted; `REMOVE_UNTAINTED_ADDR(p)` removes `p` from the set; `UNITATINED(p)` is a Boolean function checking whether `p` is in the set or not. The security specification `VC(8)` can be enforced by an assertion "`assert (!(*p==110 && state != 1 && !UNTAINTED(ap)))`". With this assertion, there is no possibility of pointer taintedness in `Vfprintf()`, and thus no possibility of format string attacks.

To correctly call the safe version of `Vfprintf()`, the compiler should insert a pre-call-stub and a post-call-stub to maintain the set of untainted addresses. In the example, because the first `Printf()` call in `main()` has a "`&j`" argument, which is an address generated by at compile time. The word "`&j`" is located in the address `stack_top-4`, so the compiler should add `stack_top-4` to the set of untainted addresses before calling `Printf()`. When `Printf()`returns, the address `stack_top-4` needs to be removed from the set.

When the above program is running, the first `Printf()` behaves as normal. The second `Printf()` behaves as normal for most input `buf`, but when the user input is an attack string, e.g., "`%d%d%d%d%d%d%d12345%n`", the program raises an exception, which demonstrates the precision of the detection technique.

7.3.2. Function `Free()`

Heap corruption attacks are due to the taintedness of free-chunk doubly-linked list. The tainted pointers are manipulated in function `free()` of some heap management systems. A heap management system is implemented, and the technique is applied on the function `Free()`, which is the deallocation function similar to the standard LibC function `free()`.

Static Extraction of Security Specification. The source code of `Free()` (shown in Table 38) is compiled to the object code (shown in Table 39), where `inst(2), inst(24), inst(25), inst(29)` and `inst(30)` are the only indirect write instructions. Starting from "`VC(2) = TAINTEDNESS(^ FreedBlock +4) is false`", the VC generator derives "`VC(1) = TAINTEDNESS(^ p) is false`". Starting from "`VC(24) = TAINTEDNESS (^ (^ BuddyBlock + 8) + 12) is false`" and "`VC(25) = TAINTEDNESS (^ (^ BuddyBlock + 12) + 8) is false`", the VC generator derives "`VC(1)= (TAINTEDNESS(^ (^ x + 8)) is false) && (TAINTEDNESS(^ (^ x + 12)) is false)`" where `x = HEAP_BASE + ((^ p) – 16 – HEAP_BASE) xor (^ ((^ p) – 16))`. Similarly, starting from "`VC(29) = TAINTEDNESS(^ FreedBlock +4) is false`" and "`VC(30) = TAINTEDNESS(^ FreedBlock +4) is false`", the VC generator derives "`VC(1) = TAINTEDNESS(^ p) is false`". Therefore, the security specification of function `Free()` is the conjunction of all these forms of `VC(1)`, i.e., "`(TAINTEDNESS(^ p) is false) && (TAINTEDNESS(^ (^ x + 8)) is false) && (TAINTEDNESS(^ (^ x + 12)) is false)`". The informal interpretation of this specification is straightforward – to securely call `Free(p)`, none of `p`, `x->Fwd` or `x->Bak` can be tainted, where `x` is the buddy block of `p`.

Table 38: C Source Code of `Free()`

```
typedef struct _HEAP_BLOCK {
    int  Size; // The size of the block.
    int  Busy; // Is this block busy?
    struct _HEAP_BLOCK * Fwd,* Bak;
} HEAP_BLOCK, * PHEAP_BLOCK;

char * BlockSizes;
void Free(char * p)
{
 int        BlockSize,i;
 char * BuddyBlock,* FreedBlock;
 int  FreeBlockListIndex, MergeExit;

 FreedBlock=p-sizeof(HEAP_BLOCK);

 // Mark this block free.
 FreedBlock->Busy=0;
 BlockSize=FreedBlock->Size;

 FreeBlockListIndex =
 CalculateFreeBlockListIndex(BlockSize);
 FreeBlockListIndex=0;
 while (BlockSize >
    *(BlockSizes+FreeBlockListIndex)) {
     BlockSize = BlockSize / 2;
     FreeBlockListIndex++;
 }
MergeExit=0;

// see if we can merge the buddy block
while (FreeBlockListIndex < 6 && MergeExit==0) {
 BuddyBlock = HEAP_BASE + (FreedBlock –
              HEAP_BASE) ^ BlockSize;
 if (BuddyBlock->Busy || BuddyBlock->Size !=
                         BlockSize)
     MergeExit=1;
 else {
  // Make a bigger block and free it.
  BlockSize*=2;
  FreeBlockListIndex++;
  if (BuddyBlock<FreedBlock)
              FreedBlock = BuddyBlock;
  BuddyBlock->Fwd->Bak=BuddyBlock->Bak;
  BuddyBlock->Bak->Fwd=BuddyBlock->Fwd;
 }
}

//Let's insert this freed block to the list.
FreedBlock->Size = BlockSize;
FreedBlock->Busy = 0;
InsertTailList(FreeBlockListIndex, FreedBlock);
}
```

Table 39: Object Code of `Free()` in Language L

```
inst(1) = mov  [FreedBlock] <- (^ p - 16)
inst(2) = mov  [^ FreedBlock + 4] <- 0
inst(3) = mov  [BlockSize] <- ^ ((^ FreedBlock + 0))
inst(4) = mov  [FreeBlockListIndex] <- 0
inst(5) = no-op
inst(6) =     branch (~(^ ((^ BlockSizes + ^ FreeBlockListIndex)) < ^ BlockSize))  10
inst(7) =     mov  [BlockSize] <- (^ BlockSize / 2)
inst(8) =     mov  [FreeBlockListIndex] <- (^ FreeBlockListIndex) + 1
inst(9) =     branch true 5
inst(10) = no-op
inst(11) = mov  [MergeExit] <- 0
inst(12) = no-op
inst(13) =     branch (~(^ FreeBlockListIndex < 6 && ^ MergeExit is 0)) 28
inst(14) =     mov  [BuddyBlock] <- ((HEAP_BASE + ((((^ FreedBlock - HEAP_BASE)) xor
                                ^ BlockSize))))
inst(15) =         branch (~(~(^ ((^ BuddyBlock + 4)) is 0) || ~(^ ((^ BuddyBlock + 0))
                                                          is ^ BlockSize))) 18.
inst(16) =         mov  [MergeExit] <- 1
inst(17) =         branch true 26
inst(18) =         no-op
inst(19) =         mov  [BlockSize] <- 2
inst(20) =         mov  [FreeBlockListIndex] <- (^ FreeBlockListIndex) + 1
inst(21) =             branch (~(^ BuddyBlock < ^ FreedBlock)) 23
inst(22) =             mov  [FreedBlock] <- ^ BuddyBlock
inst(23) =         no-op
inst(24) =         mov  [^ (^ BuddyBlock + 8) + 12] <- ^ (^ BuddyBlock + 12)
inst(25) =         mov  [^ (^ BuddyBlock + 12) + 8] <- ^ (^ BuddyBlock + 8)
inst(26) =     no-op
inst(27) =     branch true 12
inst(28) = no-op
inst(29) = mov  [^ FreedBlock + 0] <- ^ BlockSize
inst(30) = mov  [^ FreedBlock + 4] <- 0
inst(31) = no-op
```

Runtime Checking the Security Specification. The formal analysis of the source code of `Free()` clearly suggests the above security specification. A simple runtime assertion can be formulated to check this specification in practice. Given a pointer `p` to be freed, calculate `p`'s buddy block `x` is calculated. The assertion is "`x->Fwd->Bak = x && x->Bak->Fwd = x`". For an uncorrupted heap structure, since `x` is a memory block in a doubly-linked list, this assertion should holds, so there is no false positive scenario. In the meanwhile, it provides a very effective detection against heap corruption attacks. Theoretically, there could be two false negative scenarios in which the attack occurs without being detected, but these scenarios are either succeed with negligible probability, or they can only overwrite attacker-already-controllable memory locations (i.e., a chicken-and-egg situation), and thus have little use to compromise security. Let us discuss these scenarios in detail.

Scenario (1): If the values of `x->Fwd->Bak` and `x->Bak->Fwd` can be overwritten by the attacker, he/she can bypass our check by specifying `x` as the value. However, the consequence of a heap corruption attack is to overwrite the memory locations `x->Fwd->Bak` and `x->Bak->Fwd`. If the attacker has already possessed the capability of overwriting these two words, there is no need for the attack.

Scenario (2): If the input pointer `p` is a tainted value, the pointer x derived from `p` is also tainted. With a very small probability, the memory locations that the attacker wants to overwrite happen to satisfy the assertion "`x->Fwd->Bak = x && x->Bak->Fwd = x`" (again, note that the attacker attempts to overwrite `x->Fwd->Bak` and `x->Bak->Fwd`). We argue that the coincidence of this small probability with the programming bug of freeing a taintable pointer is negligible in reality.

Table 40 shows a runtime assertion embedded in `Free()`, which deploys the above assertion at the entry of the function. Function `main()` emulates a heap corruption attack. By overwriting the addresses `p+60` and `p+56`, the attack attempts to overwrite the function pointer `f` to `p`. This attack is detected by the assertion.

Table 40: `Free()` with a Runtime Assertion

```
void Free(char * p)
{  HEAP_BLOCK * x=(HEAP_BLOCK*)
                (HEAP_BASE + (((p-16) - HEAP_BASE) ^ (*(UINT*)(p-16))));
    assert( x->Fwd->Bak== x  && x->Bak->Fwd == x);

    … …(the original source code)
}
int main()
{   char * p;
    void (*f)();
    p = Malloc(40);
    *(UINT*)(p+60)=(UINT)p;
    *(UINT*)(p+56)=((UINT)&f)-12;
    Free(p);
}
```

7.4. Limitations

The two examples demonstrate that the proposed technique is able to formally derive security specifications for non-trivial programs. For these two particular examples, the extraction of security specifications is accomplished fully automatically. However, specification extraction in general is a very hard task, especially when constructing loop invariants (an induction task). Applying automatic formal techniques to analyze program properties is a focused topic in formal method and programming language communities.

The emphasis of this thesis work is primarily on the concept of pointer taintedness, rather than the automation of the formal analysis technique. The goal is to show that: (1) pointer taintedness is a common root cause of many security vulnerabilities; (2) the semantics of pointer taintedness can be formally defined; and (3) reasoning about pointer taintedness can extract security specifications, and formal methods can substantially assist the reasoning process.

CHAPTER 8

CONCLUSIONS AND FUTURE DIRECTIONS

Measurement and analysis of security vulnerabilities is crucial for designing secure computer systems. This thesis begins with the study of vulnerability reports published in *Bugtraq* list and CERT advisories. An in-depth analysis of vulnerability reports and the corresponding source code of the applications suggest three characteristics of security vulnerabilities: (1) exploits must pass through a series of elementary activities; (2) exploiting a vulnerability involves multiple vulnerable operations on several objects; and (3) the vulnerability data and corresponding code inspections allow us to derive a predicate for each elementary activity, and a security vulnerability is the result of violating the predicate in implementation. These three observations motivate the development of the FSM model to depict and reason about security vulnerabilities. Each vulnerability is modeled as a series of primitive FSMs (pFSMs), which depicts a derived predicate. The proposed FSM methodology is exemplified by analyzing several types of vulnerabilities, such as buffer overflow and signed integer overflow. The pFSMs are classified into three types, indicating three common causes of the modeled vulnerability. These causes reflect different aspects of security considerations, and suggest opportunities for providing appropriate checks to protect the systems.

Although most memory corruption attacks and Internet worms employ control-data attacks, an in-depth source code analysis of many real-world memory vulnerabilities indicates that many types of security-critical non-control data can be corrupted in order to compromise security. Our *Applicability Claim of Non-Control-Data Attacks* states that *many real-world software applications are susceptible to attacks that do not hijack program control flow, and the severity of the resulting security compromises is equivalent to that of control-data attacks.* To validate the claim, several non-control-data attacks are constructed to get the root privilege on real FTP, SSH, Telnet, and HTTP servers. Each attack exploits a different type of memory vulnerability, such as stack buffer overflow, heap corruption, integer overflow, and format string vulnerability. Based on the results of the experiments, it is shown that control flow integrity is not sufficient to

provide software security. The general applicability of non-control-data attacks represents a realistic threat to be considered seriously in defense research. The analysis shows the necessity of further research on defenses against memory corruption based attacks. Finding a generic and secure way to defeat memory corruption attacks is still an open problem. Despite their general applicability, non-control-data attacks are less straightforward to construct than are control-data attacks, because the former require semantic knowledge about target applications. Another important constraint is the lifetime of security-critical data. Reducing data lifetime is suggested as a secure programming practice that increases software resilience to attacks.

To develop comprehensive defensive techniques to defeat both control-data attacks and non-control-data attacks, it is important to extract the common root cause of most memory corruption attacks. The analysis in this thesis of various security vulnerabilities indicates that such a root cause is pointer taintedness i.e., any programming error causing a pointer value to be derived directly or indirectly from user input. By preventing pointer taintendess, many attacks can be foiled, inlucding stack smashing (taintedness of a return address), heap corruption (taintedness of the doubly-linked list of heap free chunks), format string attack (taintedness of an argument pointer) and globbing attack (taintedness of a pointer to be freed). Based on the notion of pointer taintedness, this thesis has demonstrated two techniques to enhance the security of real-world systems:

(1) Pointer taintedness avoidance: – uncovering security vulnerabilities by source code analysis. To perform source code analysis, first a formal semantic definition of pointer taintedness is given, including a taintedness-aware memory model and instruction semantics. On top of the semantic definition, a theorem-proving technique is developed to analyze C source code at machine code level. For each analyzed function, the theorem prover helps to extract a set of security specifications. The satisfaction of these specifications guarantees no possibility of pointer taintedness inside this function. This technique has been applied to analyze several LibC functions and socket read functions of HTTP servers. In each case, the negations of extracted specifications suggest scenarios of potential vulnerabilities, and thus the notion of pointer taintedness provides a unifying perspective for reasoning about security vulnerabilities.

(2) Pointer taintedness detection: checking pointers at runtime. A processor architecture level mechanism is proposed to detect pointer taintedness at runtime. Its prototype is currently implemented on the *SimpleScalar* processor simulator. The mechanism consists of four major components: taintedness-aware memory model, taintedness initialization, taintedness tracking and pointer taintedness detection. Based on an extensive evaluation using both synthetic and real-world network applications, and the SPEC benchmarks, we conclude the following: (1) the proposed architecture provides a substantial improvement in security coverage; (2) a near-zero false positive rate can be expected when the architecture is deployed; and (3) despite some synthetic false negative scenarios, running programs on the proposed architecture minimizes the chances of a successful attack; the incurred architectural overhead is likely to be low; and the approach is transparent to existing applications, i.e., applications can run without recompilation.

Future Directions. A unique aspect of this thesis research is its analysis-centric approach. A significant amount of effort is on the analysis of real-world security vulnerabilities and attacks, which has provided the basis for several new observations and ideas.

Some directions for future work include:

(1) An analysis of how different programming styles affect susceptibility to security attacks. For example, as seen in CHAPTER 4, the long lifetime of security data is a requirement of many security attacks, but in many cases this lifetime can be greatly shortened by using a different programming style. It would be interesting to explore whether compiler techniques can be used for such program style transformations.

(2) Although memory vulnerabilities are still primary threats for security, many other categories of vulnerabilities are emerging, especially those in HTTP servers and web browsers, such as cross-site scripting, cross-domain issues, SQL injection, and URL obfuscations. None of them is due to memory corruption vulnerabilities. My preliminary investigation about these vulnerabilities suggests that it is promising to apply formal semantic analysis to reason about them. A web browser enhanced to detect these vulnerabilities would benefit millions of users.

(3) To study historical data about how security vulnerabilities were discovered, reported, and patched in order to evaluate is the effectiveness of the current patching mechanism against real-world attacks, such as viruses, worms, and rootkits (i.e., kernel model malcode). An analysis of such data can motivate a more efficient way for applying patches in a more secure, more comprehensive, and more timely manner.

REFERENCES

[1] R. P. Abbott, J. S. Chin, J. E. Donnelley. *Security Analysis and Enhancement of Computer Operating Systems*. NBSIR 76-1041, Institute for Computer Sciences and Technology, National Bureau of Standards, Apr. 1976.

[2] Aleph One. Smashing the stack for fun and profit. *Phrack Magazine*, 49(7), Nov. 1996.

[3] S. Andersen and V. Abella. Data Execution Prevention. Changes to Functionality in Microsoft Windows XP Service Pack 2, Part 3: Memory Protection Technologies. http://www.microsoft.com/technet/prodtechnol/winxppro/maintain/sp2mempr.mspx

[4] Anonymous. "Once upon a free()." *Phrack Magazine*, 57(9), Aug. 2001.

[5] The Apache Software Foundation. http://www.apache.org/

[6] T. Aslam, I. Krsul, E. Spafford. Use of A Taxonomy of Security Faults. *Proc. 19th NIST-NCSC National Information Systems Security Conference.*

[7] A. Baratloo, T. Tsai, and N. Singh, Transparent run-rime defense against stack smashing attacks, In *Proceedings of USENIX Annual Technical Conference*, June 2000.

[8] D. E. Bell and L. J. LaPadula. Secure computer systems: A mathematical model Technical report MTR-2547 Vol II. Mitre Corporation, Bedford, MA, May 1973.

[9] S. Bhatkar, D. DuVarney, and R. Sekar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *Proceedings of 12th USENIX Security Symposium*. Washington, DC, August 2003.

[10] B. Bisbey II and D. Hollingsworth. *Protection Analysis Project Final Report*. ISI/RR-78-13, DTIC AD A056816, USC/Information Sciences Institute, May 1978

[11] M. Bishop and D. Bailey, *A Critical Analysis of Vulnerability Taxonomies*, Technical Report 96-11, Department of Computer Science, University of California at Davis (Sep. 1996).

[12] D. Boneh, R. A DeMillo, and R. Lipton. On the importance of eliminating errors in cryptographic computations. In *Proceedings of Advances in Cryptology: Eurocrypt '97*, pp.37-51, 1997.

[13] D. Burger and T. Austin. The SimpleScalar Tool Set, Version 2.0.

[14] Bugtraq list. http://www.securityfocus.com/bid

[15] CERT Advisory CA-2001-21 Buffer Overflow in telnetd. http://www.cert.org/advisories/CA-2001-21.html

[16] CERT Advisory CA-2001-33 Multiple Vulnerabilities in WU-FTPD, 2001.

[17] CERT Security Advisories. http://www.cert.org/advisories/

[18] S. Chen, K. Pattabiraman, Z. Kalbarczyk, R. K. Iyer. Formal Reasoning of Various Categories of Widely Exploited Security Vulnerabilities By Pointer Taintedness Semantics (Full Version). http://www.crhc.uiuc.edu/~shuochen/ pointer_taintedness

[19] S. Chen, J. Xu, R. K. Iyer, and K. Whisnant. Modeling and analyzing the security threat of firewall data corruption caused by instruction transient errors, In *Proceedings of* the *IEEE International Conf. on Dependable Systems and Networks*, Washington DC, June 2002.

[20] S. Chen, Z. Kalbarczyk, J. Xu, R. Iyer. Finite State Machine Models of Security Vulnerabilities. *http://ww.crhc.uiuc.edu/ ~shuochen/data-model-full.pdf*

[21] B. Chess. Improving Computer Security Using Extended Static Checking. IEEE Symposium on Security and Privacy, 2002.

[22] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer and C. Talcott *The Maude 2.0 System.* In Proc. Rewriting Techniques and Applications, 2003, 2003.

[23] C. Cowan, M. Barringer, S. Beattie, and G. Kroah-Hartman. FormatGuard: Automatic protection from printf format string vulnerabilities. *In Proceedings of the 10th USENIX Security Symposium*, Washington, DC, August 2001.

[24] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. PointGuard: Protecting pointers from buffer overflow vulnerabilities. In *Proceedings of the 12th USENIX Security Symposium.* Washington, DC, August 2003.

[25] C. Cowan, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. Automatic detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Symposium*, San Antonio, TX, January 1998.

[26] J. R. Crandall and F. T. Chong. Minos: Control data attack prevention orthogonal to memory model. To appear in *Proceedings of the 37th International Symposium on Microarchitecture*. Portland, OR. December 2004.

[27] D. Dhurjati, S. Kowshik, V. Adve and C. Lattner: Memory Safety without Runtime Checks or Garbage Collection. LCTES 2003.

[28] D. Evans and D. Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software*, Jan/Feb 2002.

[29] H. Feng, O. Kolesnikov, P. Fogla, W. Lee and W. Gong. Anomaly detection using call stack information. In *Proceedings of the 2003 IEEE Symposium on Security and Privacy*, May 2003.

[30] H. Feng, J. Giffin, Y. Huang, S. Jha, W. Lee, and B. Miller. Formalizing sensitivity in static analysis for intrusion detection. In *Proceedings of the 2004 IEEE Symposium on Security and Privacy*, May 2004.

[31] S. Forrest, S. Hofmeyr, A. Somayaji, and T. Longsta. A sense of self for Unix processes. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, May 1996.

[32] D. Gao, M. Reiter, and D. Song. Gray-box extraction of execution graphs for anomaly detection. In *Proceedings of the 11th ACM Conference on Computer and Communication Security*, October 2004.

[33] GHTTPD Log() Function Buffer Overflow Vulnerability. http://www.securityfocus.com/bid/5960

[34] J. Giffin, S. Jha, and B. Miller. Efficient context-sensitive intrusion detection. In *Proceedinsg of the Symposium on Network and Distributed System Security*, February 2004.

[35] J. A. Goguen and G. Malcolm. Algebraic Semantics of Imperative Programs. MIT Press, 1996, ISBN 0-262-07172-X.

[36] S. Govindavajhala and A. W. Appel. Using memory errors to attack a virtual machine. In *Proceedings* of *IEEE Symposium on Security and Privacy,* 2003, Oakland, California. May 2003.

[37] S. A. Hofmeyr, S. Forrest, and A. Somayaji. Intrusion detection using sequences of system calls. *Journal of Computer Security*, 6(3), 1998.

[38] M. Howard and D. LeBlanc, *Writing Secure Code*. Microsoft Press. 2001.

[39] Introduction to equational logic. http://www.cs.cornell.edu/Info/People/gries/Logic/Equational.html

[40] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In *Proceedings of USENIX Annual Technical Conference*. Monterey, CA, June 2002.

[41] C. Kruegel, D. Mutz, F. Valeur, and G. Vigna. On the detection of anomalous system call arguments. In *Proceeding of ESORICS 2003*, October 2003.

[42] C. Landwehr, A. Bull, J. McDermott, W. Choi, *A Taxonomy of Computer Program Security Flaws, with Examples*, ACM Computing Surveys 26, no. 3 (Sep 1994).

[43] J. –C. Laprie. Dependable Computing and Fault Tolerance: Concepts and Terminology. Proc. 15th Intl Symposium on Fault-Tolerant Computing (FTCS-15), pages 2-11, June 1985.

[44] LBNL Traceroute Heap Corruption Vulnerability. http://www.securityfocus.com/bid/1739

[45] U. Lindqvist and E. Jonsson. *How to Systematically Classify Computer Security Intrusions*. In Proc. of the 1997 IEEE Symposium on Security and Privacy, pages 154-163, Oakland, CA, May 4-7, 1997.

[46] B. Madam, K. Goseva-Popstojanova. Modeling and Quantification of Security Attributes of Software Systems. Proc. 2002 IEEE Intl Conference on Dependable Systems and Networks. Pages: 505-514. June 2002.

[47] John McLean. *Security Models.* In John Marciniak edited, *Encyclopedia of Software Engineering*. Wiley Press, 1994.

[48] C. Michael, A. Ghosh. *Simple, state-based approaches to program-based anomaly detection*. ACM Transactions on Information and System Security. Pages: 203-237. Vol.5 No.3. Aug. 2002.

[49] Microsoft Security Bulletin, http://www.microsoft.com/ technet/security/

[50] Microsoft TechNet. “Changes to Functionality in Microsoft Windows XP Service Pack 2 (Part 3: Memory Protection Technologies).” http://www.microsoft.com/technet/prodtechnol/winxppro/maintain/sp2mempr.mspx.

[51] Multiple Vendor Telnetd Buffer Overflow Vulnerability. http://www.securityfocus.com/bid/3064

[52] G. C. Necula, S. McPeak, and W. Weimer. CCured: Type-safe retrofitting of legacy code. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL*), 2002.

[53] T. Newsham. Format String Attacks. http://muse.linuxmafia.org/lost+found/format-string-attacks.pdf

[54] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the 12th Annual Network and Distributed System Security Symposium (NDSS '05)*, February 2005.

[55] Null HTTPd Remote Heap Overflow Vulnerability. http://www.securityfocus.com/bid/5774 and http://www.securityfocus.com/bid/6255

[56] R. Ortalo, Y. Deswarte and M. Kaaniche, *Experimenting with Quantitative Evaluation Tools for Monitoring Operational Security*. IEEE Transactions on Software Engineering, vol. 25, no. 5, pp.633-650, Sept. 1999.

[57] PaX Address Space Layout Randomization (ASLR). http://pax.grsecurity.net/docs/aslr.txt

[58] K. Pekka and L. Kalle. SSH1 Remote Root Exploit. http://www.hut.fi/~kalyytik/hacker/ssh-crc32-exploit_ Korpinen_Lyytikainen.html. 2002.

[59] Perl Security. http://www.perldoc.com/perl5.6/pod/ perlsec.html

[60] J. Pincus and B. Baker. Mitigations for Low-level Coding Vulnerabilities: Incomparability and Limitations. http://research.microsoft.com/users/jpincus/ mitigations.pdf, 2004.

[61] O. Ruwase and M. S. Lam. A practical dynamic buffer overflow detector. In *Proceedings of the 11th Annual Network and Distributed System Security Symposium*, pages 159–169, February 2004.

[62] Securityfocus. http://www.securityfocus.com

[63] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni. A fast automaton-based method for detecting anomalous program behaviors. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, May 2001.

[64] Sendmail Debugger Arbitrary Code Execution Vulnerability. http://www.securityfocus.com/bid/3163

[65] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address space randomization. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*. Washington, DC. Oct. 2004.

[66] U. Shankar, K. Talwar, J. Foster, and D. Wagner. Detecting format string vulnerabilities with type qualifiers. In *Proceedings of the 10th USENIX Security Symposium*, 2001.

[67] O. Sheyner, J. Haines, S. Jha. *Automated generation and analysis of attack graphs.* Proc. 2002 IEEE Symposium on Security and Privacy. Page(s): 254 –265.

[68] A. Smirnov and T. Chiueh. DIRA: Automatic detection, identification and repair of control-data attacks. In *Proceedings of the 12th Network and Distributed System Security Symposium (NDSS),* San Diego, CA, February 3-4, 2005.

[69] Solar Designer. StackPatch. http://www.openwall. com/linux

[70] SSH CRC-32 Compensation Attack Detector Vulnerability. http://www.securityfocus.com/bid/2347/

[71] StackGuard Mechanism: Emsi's Vulnerability, http://www. immunix.org/StackGuard/emsi_vuln.html

[72] P. Starzetz. CRC32 SSHD Vulnerability Analysis. http://packetstormsecurity.org/0102-exploits/ssh1.crc32. txt

[73] G. Suh, J. Lee, and S. Devadas. Secure program execution via dynamic information flow tracking. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*. Boston, MA. October 2004.

[74] K. M. C. Tan, K.S. Killourhy and R.A. Maxion. Undermining an anomaly-based intrusion detection system using common exploits. *RAID*, 2002.

[75] United States Computer Emergency Readiness Team. Technical Cyber Security Alerts, http://www.us-cert.gov/cas/techalerts/

[76] D. Wagner, J. Foster, E. Brewer, and A. Aiken. A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities. Network and Distributed System Security Symposium (NDSS2000).

[77] D. Wagner and D. Dean. Intrusion detection via static analysis. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2001.

[78] D. Wagner and P. Soto. Mimicry attacks on host-based intrusion detection systems. In *Proceedings of the 9th ACM Conference on Computer and Communications Security (CCS)*, 2002.

[79] R. Wojtczuk. Defeating Solar Designer Non-executable Stack Patch. http://geek-girl.com/bugtraq, January 1998.

[80] J. Xu, S. Chen, Z. Kalbarczyk, R. K. Iyer, An experimental study of security vulnerabilities caused by errors. In *Proceedings of the IEEE International Conf. on Dependable Systems and Networks*, Göteborg, Sweden, July 01-04, 2001.

[81] J. Xu, Z. Kalbarczyk, S. Patel and R. K. Iyer. *Compiler and Architecture Support for Defense against Buffer Overflow Attacks*. *2nd Workshop on Evaluating and Architecting System Dependability (EASY)*, San Jose, CA, October 2002.

[82] J. Xu, Z. Kalbarczyk and R. K. Iyer. Transparent Runtime Randomization for Security. In Proceedings of Symposium on Reliable and Distributed Systems (SRDS), Florence, Italy, October 6-8, 2003.

[83] W. Young and J. McHugh. Coding for a believable specification to implementation mapping, In *Proceedings of the IEEE Symposium on Security and Privacy*, 1987.

[84] WU-FTPD Remote Format String Stack Overwrite Vulnerability. http://www.securityfocus.com/bid/1387

APPENDIX: PUBLICATION LIST

Refereed Conference Publications

1. S. Chen, J. Xu, E. C. Sezer, P. Gauriar and R. K. Iyer. "Non-Control-Data Attacks Are Realistic Threats," in *Proc. USENIX Security Symposium*, Baltimore, MD, August 2005.

2. S. Chen, J. Xu, N. Nakka, Z. Kalbarczyk, R. K. Iyer. "Defeating Memory Corruption Attacks via Pointer Taintedness Detection," in *Proc. IEEE International Conf. on Dependable Systems and Networks (DSN)*, Yokohama, Japan, June 28 - July 1, 2005.

3. S. Chen, J. Dunagan, C. Verbowski and Y.-M. Wang, "A Black-Box Tracing Technique to Identify Causes of Least-Privilege Incompatibilities," in *Proc. 12th Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 3-4, 2005.

4. S. Chen, K. Pattabiraman, Z. Kalbarczyk, R. K. Iyer, "Formal Reasoning of Various Categories of Widely Exploited Security Vulnerabilities Using Pointer Taintedness Semantics," in *Proc. 19th IFIP International Information Security Conference (SEC)*, Toulouse, France, August 23-26, 2004

5. S. Chen, Z. Kalbarczyk, J. Xu, R. K. Iyer. "A Data-Driven Finite State Machine Model for Analyzing Security Vulnerabilities," in *Proc. IEEE International Conf. on Dependable Systems and Networks (DSN)*, San Francesco, CA, June 22-25, 2003.

6. S. Chen, J. Xu, R. K. Iyer, K. Whisnant. "Modeling and Analyzing the Security Threat of Firewall Data Corruption Caused by Instruction Transient Errors," in *Proc. IEEE International Conf. on Dependable Systems and Networks (DSN)*, Washington D.C., June 23-26, 2002.

7. J. Xu, S. Chen, Z. Kalbarczyk, R. K. Iyer. "An Experimental Study of Security Vulnerabilities Caused by Errors," in *Proc. IEEE International Conf. on Dependable Systems and Networks (DSN)*, Göteborg, Sweden, July 01-04, 2001.

Journal Publications

1. S. Chen, J. Xu, Z. Kalbarczyk, R. K. Iyer. "Security Vulnerabilities: From Analysis to Detection and Masking Techniques," to appear in *Proceedings of the IEEE, Special Issue on Security and Cryptography*, 2005.

2. S. Chen, J. Xu, Z. Kalbarczyk, R. K. Iyer and K. Whisnant. "Modeling and Evaluating the Security Threats of Transient Errors in Firewall Software," *Performance Evaluation*, Volume 56, Issues 1-4, pp. 53-72, March 2004.

AUTHOR'S BIOGRAPHY

Shuo Chen received an M.S. degree from *Tsinghua University* in 2000 and a B.S. degree from *Peking University* in 1997. His current research interests include security and fault tolerance, with an emphasis on systems research related to the analysis of real-world security vulnerabilities, security attacks, and the impacts of software/hardware faults on security. Shuo has internship experiences in *Avaya Labs*, *Lucent Bell Labs*, and *Microsoft Research*. All internship projects are in the area of systems and networking. He is now a researcher in the *Cybersecurity and Systems Management Group* of *Microsoft Research*.