A Data-Driven Finite State Machine Model for Analyzing Security Vulnerabilities

Shuo Chen, Zbigniew Kalbarczyk, Jun Xu, Ravishankar K. Iyer
Center for Reliable and High-Performance Computing
Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
1308 W. Main Street, Urbana, IL 61801
{shuochen, kalbar, junxu, iyer}@crhc.uiuc.edu

Abstract

This paper combines an analysis of data on security vulnerabilities (published in Bugtraq database) and a focused source-code examination to develop a finite state machine (FSM) model to depict and reason about security vulnerabilities. An in-depth analysis of the vulnerability reports and the corresponding source code of the applications leads to three observations: (i) exploits must pass through multiple elementary activities, (ii) multiple vulnerable operations on several objects are involved in exploiting a vulnerability, and (iii) the vulnerability data and corresponding code inspections allow us to derive a predicate for each elementary activity.

Each predicate is represented as a primitive FSM (pFSM). Multiple pFSMs are then combined to create an FSM model 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, input validation vulnerabilities, and format string vulnerabilities. For the studied vulnerabilities, we identify three types of pFSMs, which can be used to analyze operations involved in exploiting vulnerabilities and to identify the security checks to be performed at the elementary activity level. A demonstration of the practical usefulness of the FSM modeling approach was the discovery of a new heap overflow vulnerability now published in Bugtraq.

Key words: security vulnerabilities, data analysis, finite state machine modeling.

1. Introduction

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 [8] and Lindqvist's study on intrusions [11], and (ii) providing a degree of formalism by modeling vulnerabilities and attack characteristics; representative work includes Ortalo's Markov model of UNIX vulnerabilities [17] and Sheyner's attack graph constructor [18]. This paper 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.

Using the *Bugtraq* list maintained in *Securityfocus* [13], the study first identifies leading causes of security vulnerabilities.[1] An in-depth analysis of the reported vulnerabilities shows:

- Exploits must pass through multiple *elementary activities* – at any one of which, one can foil the exploit.
- 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 predicates that need to be met to ensure security.

These observations motivate the development of an FSM 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 easily 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*

[1] *CERT* and *Bugtraq* are two of the most comprehensive databases in which security vulnerabilities are reported. We chose *Bugtraq* for this study because its vulnerability reports are better organized and more amenable to automatic processing and statistical study.

database. For the studied vulnerabilities, we identify three types of pFSMs 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 [13]. The discovery was made when modeling another, known vulnerability.

2. Related Work

There has been significant research in modeling, analysis, and classification of security problems, some of which is based on real data.

Security Models of Access Control. A number of studies [1][2][3] have proposed models for access control that satisfies certain rigorously defined *security properties*. Bell and LaPadula [1] proposed a multilevel model and formally defined a *secure system*. A summary of the state of the art is presented in [4].

Classification and statistical analysis of security vulnerabilities. Several studies have proposed classifications to abstract observed vulnerabilities into easy-to-understand classes. Representative examples include *Protection Analysis* [10], *RISOS* [9], Landwehr's taxonomy [8], Aslam's taxonomy [7], and the *Bugtraq* classification. Similarly, taxonomies for intrusions have been proposed. Examples include Lindqvist's intrusion classification [11] and the Microsoft *STRIDE* model [12]. In addition to providing taxonomies, [8] and [11] perform statistical analysis of actual vulnerability data, based on the proposed taxonomies.

Modeling security vulnerabilities and intrusions. Several studies focus on modeling attacks and intrusions with the objective of evaluating various security metrics. Michael and Ghosh [19] 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 [18], a finite state machine 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. [17] develop a Markov model to describe intruder behavior and evaluate system security in terms of METF (mean effort to failure). Madan [20] described 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.

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. This paper uses actual vulnerability data (e.g., reports) and code inspection to derive FSMs to describe simple predicates, which are used to generate FSM models. The developed FSMs allow us to reason about the existing vulnerabilities and also seem to have the potential for discovering new vulnerabilities.

3. Analysis of the *Bugtraq* Database

3.1 Statistical Analysis

As of November 30, 2002, the *Bugtraq* database included 5925 reports on software-related vulnerabilities [13]. Each vulnerability report[2] 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[3]. 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.

3.2 An In-depth Analysis of Vulnerability Reports

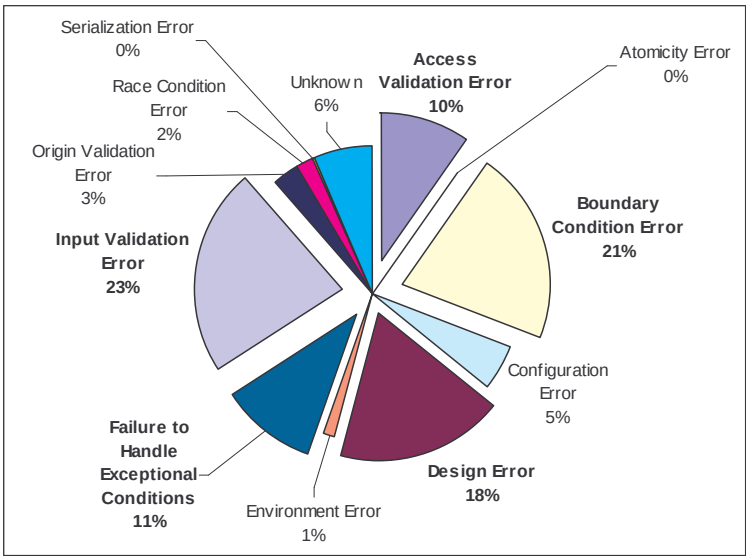
An in-depth analysis of the data and information reported in *Bugtraq* together with a close examination of the associated application code is essential to understanding the root causes of the vulnerabilities. By examining the vulnerability reports and the associated application source codes, we made 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 we model as an elementary activity) provides a security checking opportunity: failure at any one elementary activity can foil the exploit.

---

[2] Note that *Bugtraq* refers to all vulnerabilities as errors, although these may not be error in the sense defined in [6].

[3] 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

We illustrate this observation 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.

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* [15] and *split-stack* [16], 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 we will see in Section 4, 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.

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 Vulnerability*	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 Vulnerability*	A remotely supplied signed value used as an array index, allowing the corruption of a function pointer or a return address.	Execute a code referred by a function pointer or a return address	Access validation error

* #3163 denotes the vulnerability with ID 3163 in *Bugtraq*. The original information about this vulnerability can be found at http://online.securityfocus.com/bid/3163. Other *Bugtraq* vulnerabilities are also denoted in this way.

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 $0 \leq x \leq 100$.

4. State Machine Approach to Vulnerability Analysis

Our purpose in this section is to use our 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, we take three steps: (1) we 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) We model an operation on an object as a series of pFSMs. (3) We cascade the operations to model the vulnerable implementation. While our 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 us to uncover 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[4] of the function *setuid()*[5] 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 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.

---

[4] 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.

[5] The published exploit chooses *setuid()* as the target function of *GOT* entry corruption, although the targets could be other functions.

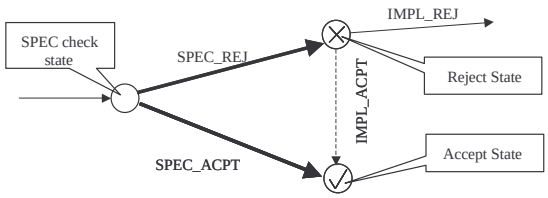


Figure 2: Primitive FSM (pFSM)

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, we associate a label *Condition*♦*Action* 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.

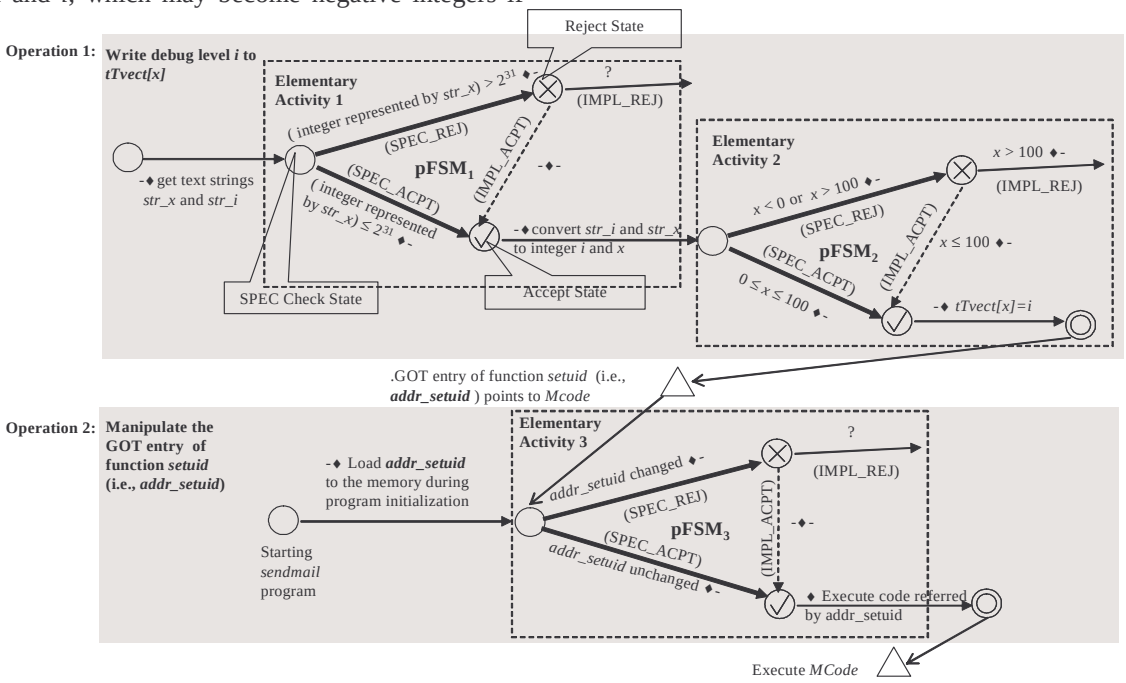


Figure 3: *Sendmail* Debugging Function Signed Integer Overflow Vulnerability

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.

5. 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.

5.1 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[6], 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 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, 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*[7].

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

[7] 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*.

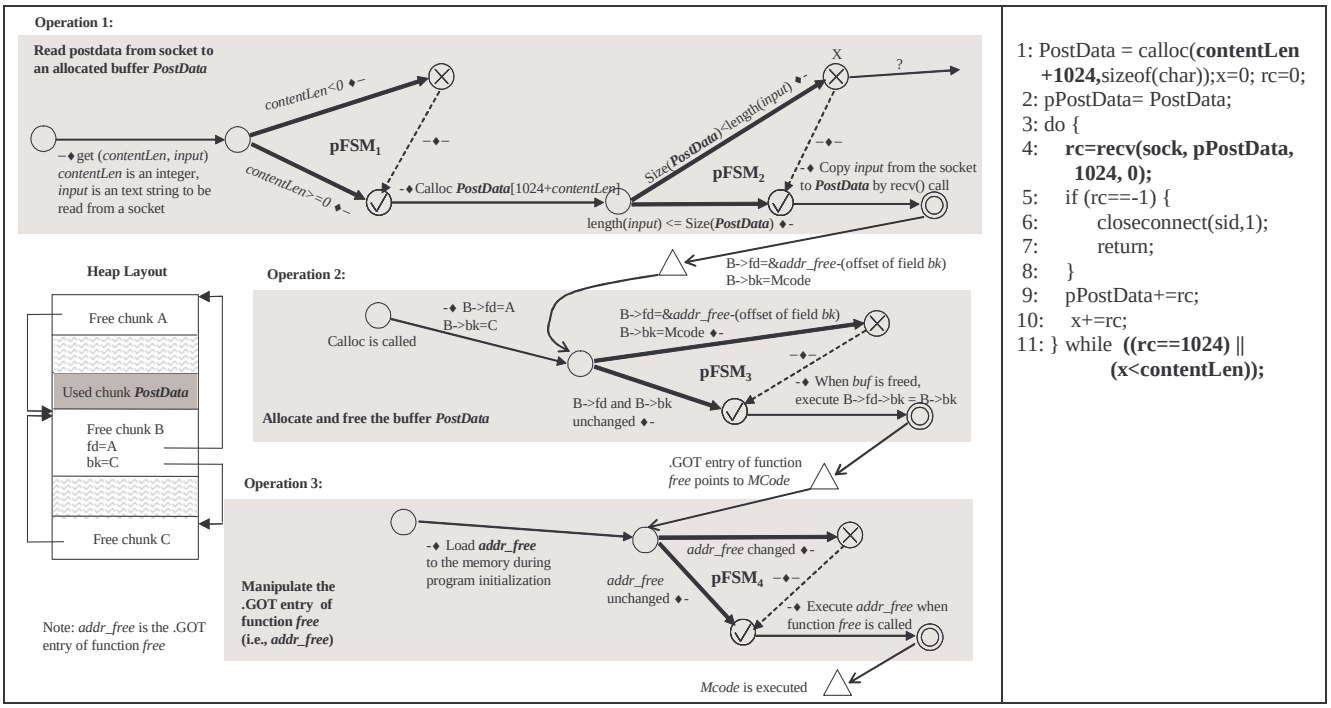


Figure 4: a) *NULL HTTPD* Heap Overflow Vulnerabilities b) Source Code, Reading *input*

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 4. 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.

5.2 Example 2: *xterm* Log File Race Condition

The program *xterm* emulates a terminal under the X11 window system. A file race-condition[8] exists when *xterm* writes messages to the user log file [1]. 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.

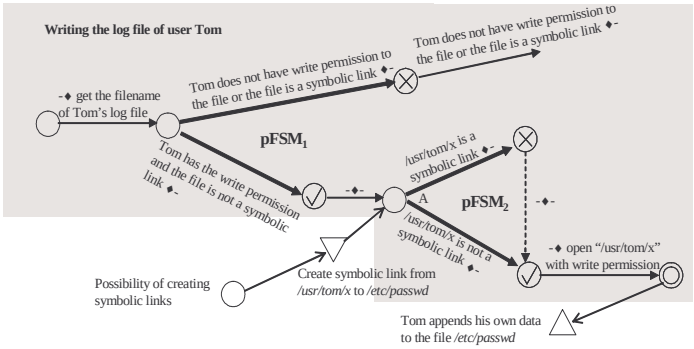


Figure 5: *xterm* Log File Race Condition

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

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

5.3 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 [8] and CA-1994-06 in [14]). 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*.

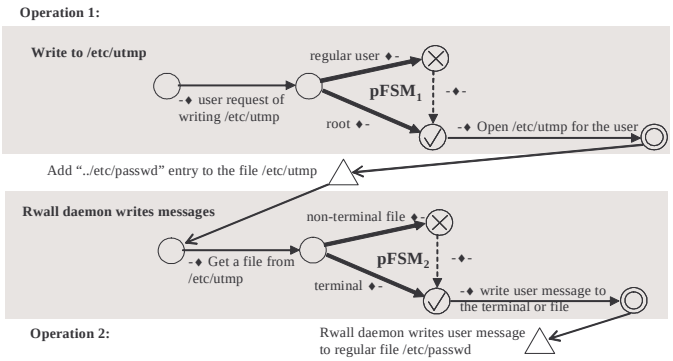


Figure 6: Solaris *Rwall* Arbitrary File Corruption Vulnerability

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*.

5.4 Example 4: Validation Error due to *IIS* Decoding Filenames Superfluously after Applying Security Checks

CGI (Common Gateway Interface) programs under the directory */wwwroot/scripts* are by design executable through HTTP request from a user. When *IIS*[9] 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 "*../*"[10], 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.

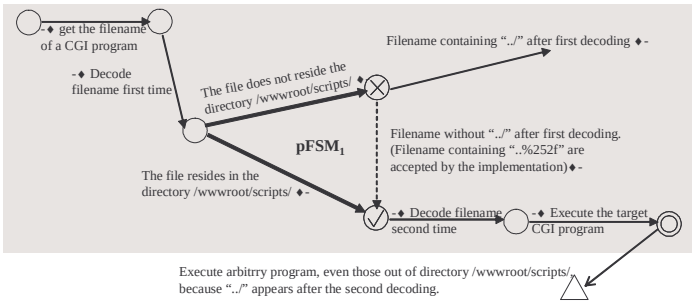


Figure 7: *IIS* Decodes Filenames Superfluously after Applying Security Checks

A Stack Buffer Overflow Vulnerability and A Format String Vulnerability. FSM is also used to model a stack buffer overflow vulnerability (*#5960: GHTTPD Log() Function Buffer Overflow Vulnerability*) and a format string vulnerability *(#1480 Multiple Linux Vendor rpc.statd Remote Format String Vulnerability)*. Due to the space limitation, we do not present the analysis of these two examples. The details can be found in [21].

6. Common Types of pFSMs

Examples in the previous sections show that the FSM approach enables a detailed modeling/analysis of several

[9] *IIS* is Microsoft Internet Information Service.

[10] 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.

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? Our 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 [20], 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 8, 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). Having defined the three types of predicates, the following lemma is stated. The proof is straightforward and is given in [21].

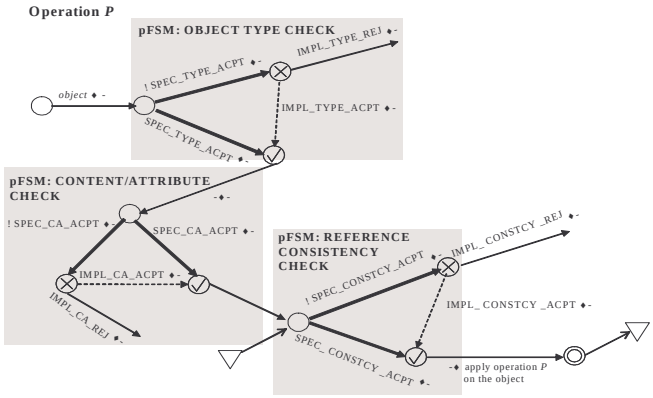


Figure 8: Types of Generic pFSMs

Lemma: (1) To *ensure the security of an operation* requires predicates (represented by pFSMs) constituting the operation to be correctly implemented. (2) To *foil an exploit* consisting of a sequence of vulnerable operations, it is sufficient to ensure security of one of the operations in the sequence.

Table 2: Types of pFSMs

Type of pFSM / Vulnerabilities	Object Type Check	Content and Attribute Check	Reference Consistency Check
Sendmail Signed Integer Overflow (Figure 3)	**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 (Figure 4)		**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 (Figure 6)	**pFSM$_2$** : Is the target file a terminal?	**pFSM$_1$**: Does the user have a root privilege?	
IIS Filename Decoding Vulnerability (Figure 7)		**pFSM$_1$**: Does the filename contain "../"?	
Xterm File Race Condtion (Figure 5)		**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 [21]		**pFSM$_1$**: size(*message*) ≤ 200 ?	**pFSM$_2$**: Is the return address unchanged?
rpc.statd format string vulnerability [21]		**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.

7. Conclusions

This paper presents a study of the security vulnerabilities published in *Bugtraq* database. The statistical study identifies leading categories of security vulnerabilities. An in-depth analysis of vulnerability reports and the corresponding source code of the applications reveal 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, (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.

A future direction of this work is to study the security predicates specific to different software (e.g., Internet services, administrative tools and TCP/IP implementation) in addition to the generic predicates discussed in this paper (e.g., buffer boundary and array index checks). We hope that a comprehensive understanding of these predicates will enable us to build an automatic tool for the vulnerability analysis.

Acknowledgments

This work is supported in part by a grant from Motorola Inc. as part of Motorola Center for Communications, and in part by MURI Grant N00014-01-1-0576. We thank Fran Baker for her careful reading of an early draft of this manuscript.

References

[1] 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.

[2] J. Rushby. Security Requirements Specifications: How and What? Symposium on Requirements Engineering for Information Security (SREIS), 2001

[3] John McLean. *Specifying and Modeling of Computer Security*. IEEE Computer 23(1) pp. 9-16. Jan. 1989.

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

[5] 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).

[6] 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.

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

[8] 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).

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

[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] 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.

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

[13] http://www.securityfocus.com

[14] http://www.cert.org

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

[16] 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.

[17] 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

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

[19] 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

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

[21] 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*