



Modeling and evaluating the security threats of transient errors in firewall software

Shuo Chen*, Jun Xu, Zbigniew Kalbarczyk, Ravishankar K. Iyer, Keith Whisnant

*Center for Reliable and High-Performance Computing, University of Illinois at Urbana-Champaign,
1308 W. Main Street, Urbana, IL 61801, USA*

Abstract

This paper experimentally evaluates and models the error-caused security vulnerabilities and the resulting security violations on two Linux kernel firewalls: IPChains and Netfilter. There are two major aspects to this work: to conduct extensive error injection experiments on the Linux kernel and to quantify the possibility of error-caused security violations using a Stochastic Activity Network (SAN) model. The error injection experiments show that about 2% of errors injected into the firewall code segment cause security vulnerabilities. Two types of error-caused security vulnerabilities are distinguished: temporary, which disappear when the error disappears, and permanent, which persist even after the error is removed, as long as the system is not rebooted. Results from simulating the SAN model indicate that under an error rate of 0.1 error per day during a 1-year period in a networked system protected by 20 firewalls, two machines (on the average) will experience security violations. This indicates that error-caused security vulnerabilities can be a non-negligible source of a security threat to a highly secure system. © 2003 Elsevier B.V. All rights reserved.

Keywords: Firewall software; Transient errors; SAN model; Security

1. Introduction

Security of critical systems can be compromised in a variety of ways. Recent studies have addressed the possibility of security violations due to errors. It has been demonstrated that errors can be exploited to obtain unauthorized access to secure or cryptographic systems. For example: (i) intentionally induced transient hardware errors have been successfully used to obtain confidential information from smartcards [1,20]; (ii) injected software errors have been used to identify regions of software vulnerable to potential security attacks [3]; (iii) naturally occurring hardware errors (errors occurring in an operational environment rather than deliberately induced errors) have been shown to result in security vulnerabilities in network applications such as FTP and SSH [6]. In the third case, relatively passive but illegal users can exploit the vulnerabilities, and while the likelihood of these events is small, considering the large

* Corresponding author.

E-mail addresses: shuochen@crhc.uiuc.edu (S. Chen), junxu@crhc.uiuc.edu (J. Xu), kalbar@crhc.uiuc.edu (Z. Kalbarczyk), iyer@crhc.uiuc.edu (R.K. Iyer), kwhisnan@crhc.uiuc.edu (K. Whisnant).

number of systems operating in the field, the probability of exploitation of such vulnerabilities cannot be neglected.

This paper experimentally evaluates and models the error-caused security vulnerabilities and the resulting security violations on two Linux kernel based firewall facilities: *IPChains* from kernel Version 2.2 and an enhanced replacement, *Netfilter* from kernel Version 2.4.

For clarity, we first define the terms *security vulnerability* and *security violation* in the context of this paper. A *security vulnerability* is a state of the software in which any packet (legal or illegal) can enter the system unchecked. If a hardware transient error results in putting the software in a state in which any packet can enter the system unchecked, an *error-caused security vulnerability* is said to occur. The time period during which such a vulnerability persists is called a *window of vulnerability*. If more than five malicious packets¹ enter the system during a window of vulnerability, a *security violation* is said to have occurred.

There are two major aspects to this work:

1. Extensive error injection experiments on the two firewalls to analyze how an error in the firewall code implementing security policies can compromise system security.
2. A Stochastic Activity Network (SAN) model to quantify and analyze the interaction between the error occurrence process, the packet arrival process (i.e., the arrival of legal or illegal packets to the firewall), and processor instruction cache miss rates and replacement characteristics.

The error injection experiments show the following:

- Of errors injected into the firewall code segment, about 2% cause security vulnerabilities.
- There are two distinct types of error-caused vulnerabilities:
 - A *temporary security vulnerability* results from an error that impacts an executing instruction. It disappears when the erroneous location is overwritten, cached out, or the system is rebooted. The resulting window of vulnerability can be as short as a single instruction, if only the executing instruction is affected. If the error affects the cache, the vulnerability persists until the instruction is cached out.
 - A *permanent security vulnerability* also results from an error that impacts an executing instruction, but now it corrupts the semantic or structural integrity of the permanent data structures. Removing such errors without rebooting the system does not eliminate the security vulnerability.

Using data on field failures, data from the error injection experiments, and system performance parameters such as processor instruction cache miss and replacement rates, we simulate the proposed SAN model to predict the mean time to security vulnerability and the duration of the window of vulnerability under realistic conditions. The results indicate that the error-caused vulnerabilities can be a non-negligible source of security violations. For example, under a realistic error rate of 0.1 error per day during a 1-year period in a networked system (e.g., a large university campus) protected by 20 firewalls, two machines (on the average) will experience security violations.

How does this relatively low rate of error-caused vulnerabilities compare with the rate of security vulnerabilities due to software bugs? Publicly available reports [16] on security vulnerabilities in RedHat Linux 7.0 (or above) show 13 vulnerabilities recorded during a 21-month period (November 2000–August

¹ With as few as five packets, an intruder can establish a TCP connection with the remote machine and launch an actual attack, e.g., exploiting the FrontPage Server Extension Sub-Component Buffer Overflow Vulnerability [17].

2002). Although a substantive conclusion cannot be arrived at by simply comparing these two data points, it is clear that error-caused security vulnerabilities are important in a highly secure environment.

2. Related work

A number of researchers have studied the relationship between errors and system security. Past research has focused on four issues: (1) the impact of intentionally induced hardware transients on smartcards to compromise cryptographic systems; (2) techniques for locating security-sensitive regions of applications; (3) modeling real systems to quantify the probability of a security vulnerability due to system misconfiguration or software bugs; (4) the impact of hardware errors on system security.

Several studies have shown that hardware transients can be used as an effective means to attack smart-card-based cryptographic systems to which attackers can have physical access. Boneh and DeMillo [2] found that a popular implementation of the RSA² [4,21] encryption algorithm, based on the Chinese Remainder Theorem (CRT), is vulnerable to hardware transients. Such transients occurring during certain phases of the CRT-based private key operation can cause an erroneous cipher-text to be produced. A recipient of the erroneous cipher-text can derive the RSA private key using a simple greatest common divisor (gcd) operation. Using the power analysis techniques from Kochar et al. [7], one can pinpoint different phases of cryptographic algorithms. Once the vulnerable phase is located, techniques similar to those used by Kuhn and co-workers [1] can be applied to inject transient faults to discover the secret key. Ghosh and O'Connor [3] present a software tool that uses a source-code-based mutation test to identify security-critical regions of programs that, if flawed, can result in security vulnerabilities. The authors argue that programmers can take advantages of the tool to identify security-sensitive segments. This information can also help designers conduct code inspections or formal analysis to eliminate potential sources of vulnerabilities. Research in [6] quantifies the impact of hardware errors on the FTP server and SSH server applications. The paper shows that errors can impact the authentication code of these applications in such a way that relatively passive intruders can gain access to the servers using correct logins but random passwords.

In [14], a Markov model is used to evaluate the effort required of an attacker to reach the specified target. Using an attacker's effort to create a security violation as a key model parameter, the study models several known vulnerabilities in UNIX.

The present study focuses on the impact of errors affecting the semantics and/or the structural integrity of the firewall data structures that store rules for enforcing security policies. Any corruption of these data structures can potentially create temporary or permanent security vulnerabilities. Via experimental evaluation and a SAN model, the possibility of security violations for realistic error rates, processor/cache performance parameters, and packet arrival rates (legal or illegal) is evaluated.

3. Target applications: *IPChains* and *Netfilter* firewalls

A *firewall* is a mechanism for enforcing network security policies; it keeps unauthorized packets from gaining access to the machines it protects. It also prevents insiders from accidentally exposing confidential

² The RSA cryptosystem is a public key cryptosystem that offers both encryption and digital signatures (authentication) [14].

information (e.g., network topology) or critical services. The core functionality of a firewall is *packet filtering*: filtering out potentially malicious or unauthorized network packets while letting legitimate packets through. Typically, a network administrator specifies a set of rules and policies (e.g., “reject any packet from *enemy.com*”) that the firewall implements.

This paper studies two widely used firewalls: *IPChains* [8] and *Netfilter* [23]. *IPChains* is implemented in Linux kernel Version 2.2; its sole function is packet filtering. *Netfilter*, built into Linux kernel Version 2.4, provides additional features, including Network Address Translation, which converts an IP address into another IP address and is used for IP masquerading and load balancing, as well as stateful packet inspection, which maintains information on the state of each connection, and enables checking rules that use state information at a finer level of granularity. Thus, one policy can be applied to the first packet of a connection and another to packets belonging to an established connection.

3.1. Structure and packet filtering of IPChains

Rules in *IPChains* are organized as a linked list of *rule-chains*. Each rule-chain is structured as a linked list of *rules* and a *policy* for accepting or rejecting an incoming packet. Fig. 1 illustrates an example configuration of *IPChains* that consists of two rule-chains. Each rule in a rule-chain has a *condition* (e.g., source address is *a.b.c.d*) and a *branch target*, which specifies the next rule-chain to be traversed (depicted by a solid arrow in Fig. 1) if a packet matches the specified condition. Starting with *Rule-chain1*, a packet traverses the linked list of rules. If the packet matches the condition of a rule, the traversal jumps to the rule-chain in the branch target field. The traversal stops when the packet either reaches the end of a rule-chain without a match or reaches a NULL rule-chain, containing only a defined policy. In either case, the policy of the rule-chain (accept/reject) is applied to the packet. In Fig. 1, the dashed line depicts the path traversed by *IPChains* for a packet from address *a.b.c.d* destined to address *w.x.y.z*.

The two critical functions that implement the packet filtering are *ip_fw_check* and *ip_rule_match*. Function *ip_fw_check* takes the header of an input packet and implements the rule-chains traversal algorithm of Fig. 1. Function *ip_rule_match* is invoked by *ip_fw_check* to determine whether a packet matches a rule.

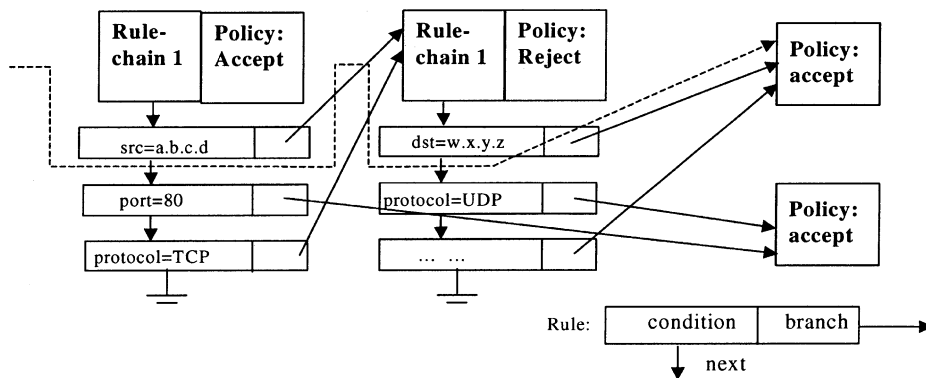


Fig. 1. Rule-chain structure in *IPChains*.

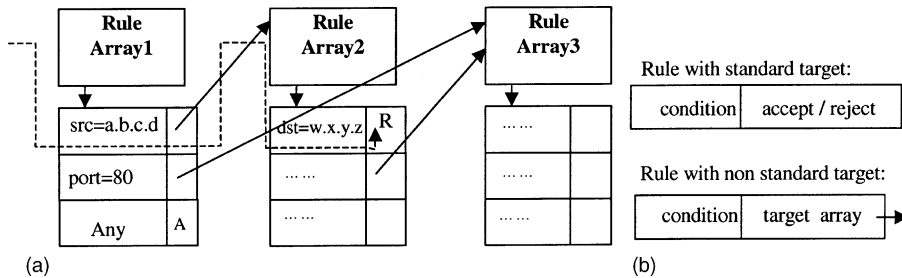


Fig. 2. (a) Rule array structure in *Netfilter*. (b) Two kinds of rules in *Netfilter*.

3.2. Structure and algorithm of *Netfilter*

Rules in *Netfilter* are stored in *rule arrays* rather than in linked list rule-chains (as in *IPChains*). Policies are associated with rules rather than with rule-chains. Each rule has a condition field so that the target field now defines the action in the case of a condition match. Two possible actions can be specified in the target field: a *policy* and a *target array*. When a packet matches the condition in a rule and if a policy is associated with the rule, it is applied; otherwise, the algorithm jumps to the specified rule array. Fig. 2a shows an example of a *Netfilter* configuration containing a set of rules and policies similar to those in Fig. 1. The dashed line depicts the path taken by *Netfilter* for the case of the same packet illustrated in Fig. 1. *Netfilter* exits the rule checking only when a rule matches the packet, in which case the policy specified for this rule is applied (the packet is rejected (R) in the example in Fig. 2).

Two critical functions from *Netfilter* were selected as targets for this study: *ipt_do_table* and *ip_conntrack_in*. The former implements the traversal algorithm; the latter is the function performing the stateful inspection.

4. Experimental setup and approach

The goal of the experimentation was to determine the probability that an error impacting the semantics or structure of the firewall data will result in temporary or permanent security vulnerability. Toward this end, several machines were setup in pairs. Each pair consisted of a firewall machine and an attacker machine whose sole purpose was to bombard the firewall with illegal packets. Errors were injected exhaustively into the four specified rule checking functions for the two firewalls. Each run of the error injection experiment involved a single-bit flip in an instruction, while a packet was sent from the attacker machine to expose the impact of the error on firewall. The specific steps of an injection run are described below and illustrated in Fig. 3:

1. Identify a target address in the text-segment of the firewall code, and identify a bit in the targeted text-segment for injection.
2. Flush a *begin* record to the experiment data-log file.
3. Inject the error at the target bit of the target address.
4. Send a *ping* packet from the attacking machine:

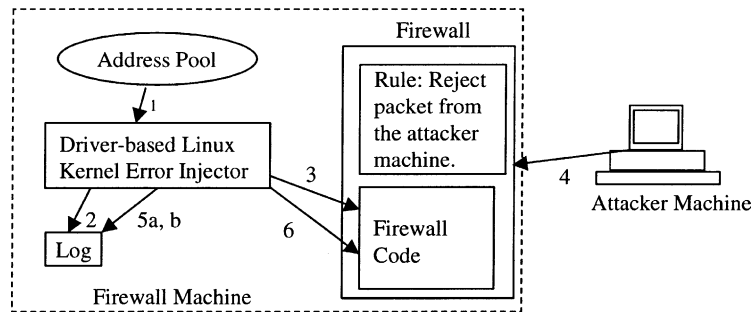


Fig. 3. Steps in a single error injection run.

- (a) If the packet is received by the firewall, flush a *violation* record to the log.
- (b) If no packet is received, flush an *ok* record to the log.
- (c) If system hangs or crashes, reboot the machine.

5. Remove the current error condition; go to Step 1.

This experiment required three elements: (1) mechanisms for injecting errors into the operating system kernel; (2) support for automating the injection process; (3) provision for data collection.

Injecting errors into the operating system kernel. A Linux driver-based error injector was developed for injecting errors into Linux kernel memory space. The injector is invoked using NFTAPE [5,22] to conduct the error injection campaign.

Automating the experiments. Errors injected into the operating system kernel usually cause the kernel to hang or crash, requiring a manual reboot. To minimize human intervention, two techniques were employed: (1) the kernel-panic exception handler was modified so that when the kernel went into a panic or crashed, the machine would auto-reboot; (2) each machine hosting the firewall was fitted with a hardware watchdog card [26] to reboot automatically in case of a hang.

Collecting the experimental data. Data from experiments were written to the log file on the local disk. The operating system usually buffers the log in the memory-I/O cache and periodically flushes the log data to the disk. If an injected error causes a system crash, chances are that the buffered data will be lost. To prevent loss of data, the results from the experiments were flushed to disk as soon as they were available. After a reboot due to a crash, the error injector control mechanism read the stored log and restarted the experiment from the point at which the data was lost.

4.1. Error model

The error model assumed in this study is an error that impacts the correct execution of an instruction by the processor. An error can originate in the disk, network, bus, memory, cache, or processor execution core. Although error detection techniques such as cyclic redundancy code (CRC), parity, and ECC are available, errors can escape such mechanisms for several reasons. First, as CMOS technology scales down to smaller feature size and lower voltages, both memory and processors are more susceptible to soft errors [24,25]. Second, error detection and correction mechanisms do not cover all critical components in a system. A study of microprocessor error detection mechanisms [9], for example, shows that only

main memory and caches are protected by ECC or parity on the widely used Intel IA-32 architecture; the processor execution core had no protection. Moreover, many network equipment vendors use memory components that do not incorporate error detection mechanisms, and equipment such as hardware routers and firewalls, once deployed, are susceptible to soft errors.

In this work, single-bit errors are injected so as to impact the instructions of the target applications (the packet filtering functions). This allows us to emulate errors in the main memory, the cache, the processor execution buffer, and the processor execution core, as well as errors occurring during the transmission over a bus. Previous research on microprocessors [10–12] has shown that most (90–99%) device-level transients can be modeled as logic-level, single-bit errors. Data on operational errors also indicates that a majority of errors in the field are single-bit errors.

4.2. Outcome categories

Outcomes from error injection experiments can be classified into following three categories:

1. *Error not activated or not manifested (NA + NM)*: The corrupted instruction is not executed or does not cause a visible, abnormal impact on the system.
2. *Crash or hang (CRASH + HANG)*: The Linux kernel raises an exception or hangs.
3. *Temporary or permanent security vulnerability (TSV or PSV)*.

Before providing the statistics on the error injection experiments, we will discuss several examples of security vulnerabilities observed in our experiments. In each case, the security vulnerability was revealed via the error injection experiment. Clearly it would be difficult, if not impossible, to identify these vulnerabilities via code inspection.

5. Examples of security vulnerabilities

5.1. Temporary security vulnerabilities

Fig. 4 shows a segment of code from *IPChains* that checks whether the source address of the incoming packet ($ip \rightarrow src_addr$) is within the source address range of the rule pointed to by f (the range is specified by $f \rightarrow src_addr$ and $f \rightarrow src_netmask$). The equivalent assembly code contains three instructions:

1. Fetch the source address of the packet.
2. *AND* with the netmask of the rule.
3. Compare with the address specified in the rule.

C code: <code>if ((ip->src_addr & f->src_netmask) == f->src_addr)</code>	
Binary and assembly:	
(1) 8b 57 0c	<code>mov 0xc(edi),edx //mov ip->src_addr, edx</code>
(2) 23 55 08	<code>and 0x8(ebp),edx //and f->src_netmask,edx</code>
(3) 3b 55 00	<code>cmp 0x0(ebp),edx //cmp f->src_addr,edx</code>

Fig. 4. Example of temporary data corruption.

If the comparison *cmp* in instruction (3) results in an equality, the packet is in the address range specified by the rule; if *cmp* yields *not-equal*, the packet is not in the range, i.e. the rule is not applicable to the packet. At this point, the checking can terminate and allow the packet in or jump to the next rule in the current chain.

A security vulnerability occurs when an error causes the condition in the *if* statement to be *false*, which causes the current rule (rule pointed to by *f*) to be skipped, even when the packet matches the rule. If, in instruction (1), byte *0c* is changed into *08* (a single-bit error), the instruction becomes *mov 0x8(edi),edx*, and as a result the register *edx* is assigned an incorrect value. Similarly, in instruction (2), if an error results in byte *23* to change to *33* (again a single-bit flip), the first instruction becomes *xor 0x8(ebp),edx*. As a result the IP address of a packet being filtered is *XOR*'ed with the netmask, rather than *AND*'ed. Both errors cause the result of the *cmp* instruction to be *not-equal*, i.e., in each case the source address of the packet is not in the source address range of the rule, so rule checking is bypassed. There are 9 bytes, or 72 bits, in the code segment described in Fig. 4. Our experiment shows that 42 bits in this 72 bit code segment can cause security vulnerabilities when flipped, i.e., allow illegal packets to penetrate the firewall.

It is clear from the above example that the described errors can result in one or more illegal packets going through the firewall. An important question to ask is how long this condition will persist, because the longer the error persists, the higher the probability that malicious packets will be allowed to enter the system. The answer depends on which component is hit by the error. There are two possibilities:

1. If the error only hits the instruction during its execution (i.e., when it is going through the instruction pipeline or the execution engine), then clearly only the current (potentially illegal) packet being checked will be allowed in. Assuming no further error propagation, the next packet will be error-free, since it is checked by a new, error-free copy of this instruction.
2. If the error hits the instruction cache (e.g., L0 cache), the instruction will continue to process incorrectly until the affected cache line is replaced. Thus the higher the cache level (or memory), the longer the duration of the error. As a last resort, the error will be removed when the system is rebooted.

5.2. Permanent security vulnerabilities

Fig. 5 shows the pseudocode from the function *ip_fw_check* in *IPChains*. It is written in a mixture of C and x86 assembly. Boxes L1, L2, and L3 are C code equivalent to the corresponding assembly segments. The pointer *FirstRule* points to the address of the first rule in the rule-chain to be examined; pointer *f* points to the rule being examined, and *ip* points to the incoming packet.

This code implements the traversal algorithm described in Section 3.1. It first initializes the rule pointer *f* and then traverses the rule-chains within a loop using *f*. Corruption of instructions within this code segment can lead to permanent data corruption. The corruptions can impact either the rule structure (structural integrity) or the rule content (semantics integrity). Rule structure corruption occurs when an error corrupts the pointers or data needed to maintain the rule structure; rule content corruption refers to the corruption of the information specifying individual rules, e.g., Source/Destination IP addresses, netmasks, or port numbers. Four cases of permanent data corruption observed in our experiments in *IPChains* and *Netfilter* are described in the following:

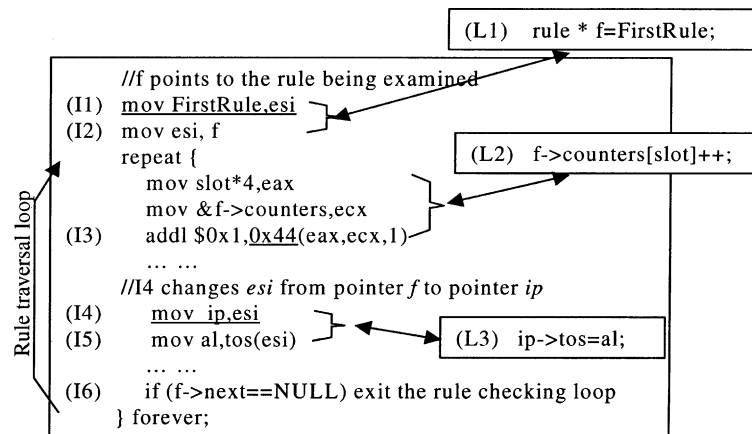


Fig. 5. Examples of permanent data corruption.

- *Case 1: Rule structure corruption.*

Instructions I1 and I2 (corresponding to L1 in Fig. 5) initialize the pointer f to be the address of the first rule in the rule-chain. An error in I1 (e.g., opcode $0x8b$ is changed into $0x89$) reverses the direction of the MOV instruction to `mov esi,FirstRule`. As a result, instruction I1 changes the value of `FirstRule`. Because the pointer `FirstRule` is on a heap, its erroneous value persists. Next, I2 assigns this erroneous value to the loop variable f , which makes $f \rightarrow next$ to be NULL. The erroneous value of f causes the loop to bypass further rule checking³ and to accept the malicious packet without system crash or hang. In this example, the corruption to the rule structure `FirstRule` pointer resulted in a permanent security vulnerability. Any newly arriving (legal or illegal) packet will go through the firewall because f is always incorrectly set and rule checking is always skipped, even after the transient error disappears.

- *Case 2: Data structure offset corruption.*

In *IPChains*, most fields in the rule data structures are intended to be read-only for the rule traversal functions. There are, however, some fields (e.g. statistics counters) that are updated during rule traversal. These counters are incremented whenever *IPChains* receives a packet. In the code segment given in Fig. 5, instruction I3 increments such a counter. An error corrupting the offset $0x44$ (offset of the counter) can effectively invalidate the rule. For example, in our experiments, an error changed the offset $0x44$ to $0x04$ (the offset of *destination IP address* field). Hence, instruction I3 (corresponding to L2 in Fig. 5) increments the *destination IP address* field by one and effectively invalidates the rule: the corrupted destination IP address field allows packets that should be rejected to get through.

- *Case 3: Mishandled pointer alternation.*

The x86 processor has a limited number of general-purpose registers, so it is very common that some variables share a given register during the execution. In the code segment given in Fig. 5, the pointer f , which points to the current rule, shares the register `esi` with the pointer `ip`, which points to the header of the incoming IP packet. It is crucial that the transition between the two is accomplished correctly. Errors in I4 can change this instruction to `mov ip,edx` (or `dh, edi`, depending on which bit is in error).

³ *IPChains* uses $f \rightarrow next == NULL$ as the exit condition from the loop (16).

As a result, the execution of I4 does not assign the value of *ip* to *esi*. Subsequent instructions that use *esi* to access *ip* effectively access *f*. For example, I5 (corresponding to L3 in Fig. 5) modifies *ip* → *tos*. Due to the error propagation from I4 above, the modification performed by I5 is applied to the rule pointed to by *f*, and the content of the register *al* is written to the *source IP address* field of the rule. This erroneous modification to the rule causes permanent security vulnerability.

- *Case 4: Direct rule content corruption.*

In the experiments with *Netfilter*, we found three instances in which an error reversing a MOV instruction leads to corruption of the rule content. An example is taken from *Netfilter* code, where the instruction *mov 0x8(ebx),eax* is used to fetch the *destination address* stored in the rule into register *eax*. Suppose an error in this instruction can reverse the direction of a MOV instruction to *mov eax, 0x8(ebx)*, which writes *eax* to the *destination address* field in the rule content. Obviously, such a corrupted rule cannot properly recognize packets that should be rejected. Observe that this is also a permanent security vulnerability.

Next we will compare the rule traversal functions of *IPChains* and *Netfilter* mainly from the perspectives of data structure, programming style, and compiler strategy.

5.3. Comparison of rule traversal functions of *IPChains* and *Netfilter*

Based on an examination of the observed security vulnerabilities, Table 1 provides a comparison of the two firewalls in terms of (1) data structure type (used to store the rules and security policies), (2) programming style (used in implementing the rule traversal algorithm), and (3) the compiler strategy employed to generate a binary image of the firewall software. The table also provides simple suggestions to alleviate the observed problems.

Table 1
Comparison of rule traversal functions of *IPChains* and *Netfilter*

Comparison	Suggestion
<i>IPChains</i> uses NULL value as exit condition, i.e., when none of the predefined rules matches the data packet, the packet is accepted by default (Case 1). <i>Netfilter</i> only accepts those packets that match a rule whose corresponding security policy is explicitly <i>ACCEPT</i> . <i>Netfilter</i> is free from the security vulnerabilities caused by rule structure corruption	Avoid using common value (e.g. NULL) as the exit condition in programming the rule traversal algorithm
Neither <i>IPChains</i> nor <i>Netfilter</i> explicitly protects the offset of a write operation (Case 2) or the content of a rule (Case 4) from being corrupted. Case 2 illustrates that by introducing the counter (which is modified every time a packet comes) in the rule structure of <i>IPChains</i> and <i>Netfilter</i> , the rules are no longer read-only	Do not mix sensitive and read-only data with potentially writable data. Use the memory protection mechanism of the operating system to protect read-only data
Case 3 illustrates security vulnerability due to corruption of an instruction handling a shared register (the register is used to preserve a pointer to the sensitive data as well as another variable alternatively). While <i>IPChains</i> suffers from such register sharing, <i>Netfilter</i> is free from the security vulnerabilities caused by mishandled pointer alternation. In the latter case, the compiler-generated binary does not share the register pointing to the rule with any other variable	The compiler should avoid sharing registers that represent pointers to sensitive data

6. Experimental results

This section presents results from error injection experiments on the firewalls *IPChains* and *Netfilter*. Recall that exhaustive error injections were performed on the packet filtering functions of these firewalls. Each bit in the code of *ip_rule_match* and *ip_fw_check* (for *IPChains*) and *ipt_do_table* and *ip_conntrack_in* (for *Netfilter*) was injected. Table 2 provides the frequency and the measured probabilities for each outcome category and for the four specified functions.

The results show that

1. Errors injected to the target functions cause a measurable number of security vulnerabilities: 369 (0.021 probability) for *IPChains* and 307 (0.020 probability) for *Netfilter*. Approximately 98% of the security vulnerabilities are temporary. Between 1 and 3% of the security vulnerabilities are permanent. Although permanent security vulnerabilities constitute a relatively small fraction of the measure of vulnerabilities, they present a significant security threat. Once they occur, the time window of vulnerability can be long, potentially allowing many malicious packets to enter the network.
2. There is significant difference in the frequency (and the probability) of the observed security vulnerabilities for the four target functions. For example, errors injected to *ip_rule_match* result in 304 (0.052 probability) vulnerabilities versus 65 (0.0054 probability) cases for *ip_fw_check*. This is due to the different functionalities of these functions. *ip_rule_match* traverses the rule_chains, trying to match the incoming packet to a rule. The traversal involves a significant number of pointer operations. Hence an error in this function is most likely to corrupt a pointer and crash the system. The function *ip_rule_match* determines whether an incoming packet matches a given rule and involves a number of logic operations. Errors in this function are more likely to cause incorrect result without crashing the system, thus creating the possibility of a security vulnerability.
3. There is a significant probability (0.20 for *IPChains* and 0.41 for *Netfilter*) that errors will lead to system crash or hang.
4. A large number of injected errors do not have a visible impact on the system. The probability that an injected error is not activated, or is activated but does not manifest, is 0.78 for *IPChains* and 0.57 for *Netfilter*.

The foregoing experimental data quantifies the probabilities that, given an error, a security vulnerability will occur. In order to analyze the likelihood of these vulnerabilities becoming real security violations, we describe a conceptual model. Using this model, a SAN model is proposed, to determine the probability

Table 2
Results for *IPChains* and *Netfilter*

Function	NA + NM	CRASH + HANG	Security vulnerability	Total	Temporary SV	Permanent SV
<i>IPChains</i>						
<i>ip_rule_match</i>	4682 (0.80)	846 (0.15)	304 (0.052)	5832	304 (0.052)	0 (0)
<i>ip_fw_check</i>	9278 (0.77)	2785 (0.23)	65 (0.0054)	12128	55 (0.0045)	10 (0.0008)
Total	13960 (0.78)	3631 (0.20)	369 (0.021)	17960	359 (0.02)	10 (0.0005)
<i>Netfilter</i>						
<i>ipt_do_table</i>	6251 (0.60)	4139 (0.40)	95 (0.0091)	10480	91 (0.0087)	4 (0.0004)
<i>ip_conntrack_in</i>	2649 (0.51)	2387 (0.45)	212 (0.04)	5248	212 (0.040)	0 (0)
Total	8900 (0.57)	6526 (0.42)	307 (0.020)	15728	303 (0.019)	4 (0.0002)

of error-caused temporary and permanent security vulnerabilities and the resulting security violations for a range of operational conditions (e.g., different packet arrival rates, varying processor utilization by the firewall and non-firewall workloads).

7. Conceptual model

Fig. 6 describes a simplified model for reasoning about the conditions under which an error can result in a security vulnerability, which, in the presence of malicious packets, results in a security violation. The model assumes that errors arrive at a specified rate (e.g. 0.1 per day). At time t_1 , an error occurs with no manifestation; it either disappears (e.g., it is overwritten) with no effect, causes undetectable effects, or becomes latent. At time t_2 , another error occurs, but this time it results in a system crash and a subsequent system reboot. At time t_3 , yet another error arrives that causes a temporary security vulnerability. This vulnerability lasts until time t_4 , when the erroneous instruction is evicted from the cache. At time t_5 , the system crashes because of a previous latent error and is rebooted again. Another error arrives at time t_6 , and this time it propagates to the firewall rule data structure and causes a permanent security vulnerability. This vulnerability lasts until either the resulting security violation is detected by an intrusion detection system or the system crashes due to a new or latent error (time t_7). Two types of windows of vulnerability occur: a window of temporary vulnerability (t_3 – t_4) and a window of permanent vulnerability (t_6 – t_7). Observe that during the entire period (t_1 – t_8), the error arrival process interacts with the arrival of data packets, some of which are generated by an intruder's machine. The illegal packets try to penetrate the firewall and compromise the system security. A security violation occurs when illegal packets (five or more) penetrate the firewall during a window of security vulnerability.

It is clear from the conceptual model that the duration of the window of security vulnerability and the probability of malicious packets getting pass the firewall (i.e., the probability of a security violation) is dependent on several factors:

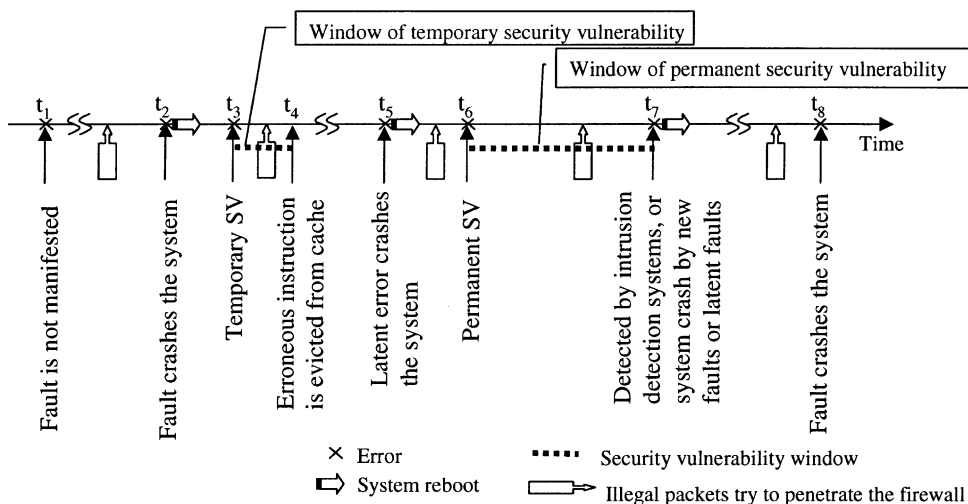


Fig. 6. Conceptual model.

1. Error arrival rate.
2. Cache parameters (cache miss rate, cache size, and fetch rate).
3. Processor utilization dedicated to packet filtering (or packet arrival rate at the firewall).
4. Other workload (non-packet filtering workload) on the firewall machine.
5. Fraction of malicious packets.

8. SAN model for security vulnerabilities

Fig. 7 shows the SAN model for detailed analysis of error-caused security vulnerabilities. The model consists of two submodels: an *error submodel*, which depicts the error behavior including error occurrence, cache fetch and replacement, processor execution and the creation of an ensuing security vulnerability, and a *workload submodel*, which depicts the workload in the system. There are two tokens in the overall model: an *error token* (initially in the *error* place) representing errors, and traveling along the error submodel and a *job token* (initially in the *job* place) representing jobs, and traveling along the workload submodel.

The jobs are divided into firewall-related tasks, non-firewall-related tasks, and idle (representing no jobs). Firewall-related tasks refer to the packet filtering activity. Non-firewall workload represents the processor executing tasks other than packet filtering; this can include other OS tasks, book-keeping tasks and non-OS-related processing. Clearly, in a dedicated machine, these tasks are not dominant. The workload submodel interacts with the error submodel via input gates *CPU working*, *firewall enable*, *non-firewall workload enable* and via an output gate, *task switch*. The three input gates enable or disable

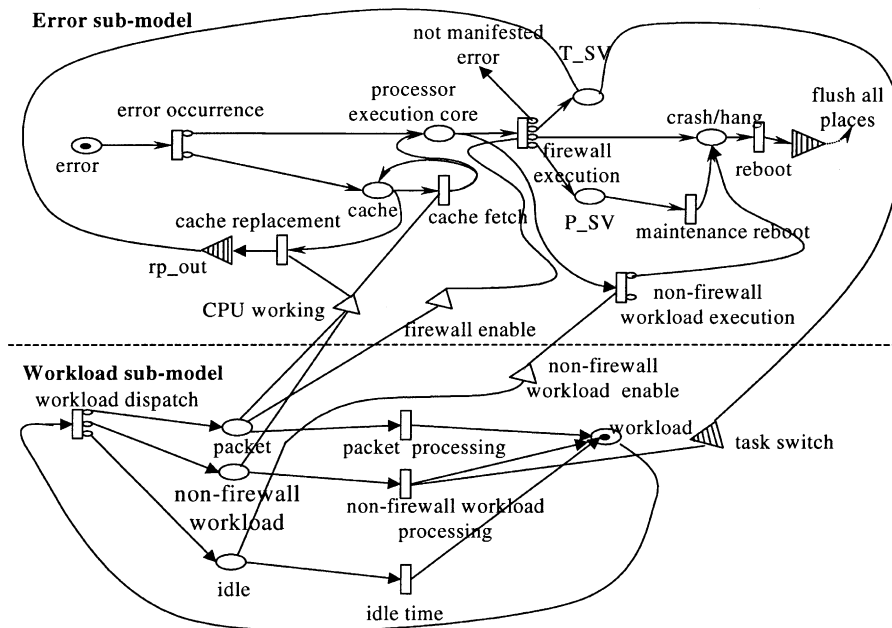


Fig. 7. SAN model for analyzing error-caused security vulnerabilities.

activities in the error submodel based on the states of the places in the workload submodel. The *task switch* gate removes tokens from the *T_SV* place during a task switch.

Workload submodel. The job token in the model represents either the invoking of the firewall activity by an arriving data packet or the arrival of non-firewall-related jobs for execution. The processor remains idle if no job is in the system. The *job dispatch* activity and an associated probability distribution determine the percentage of time the processor handles data packets, executes non-firewall workload, or remains idle.

Depending on the place where the job token resides, different activities in the error submodel are enabled. The gate *CPU working* enables the cache activity, either *cache replacement* or *cache fetch*. The *Firewall enable* and *non-firewall workload enable* gates permit the processor to either handle a packet (legal or illegal) or execute non-firewall workload.

Error submodel. An error arriving at the system can impact either the *processor execution core* or an instruction in the processor cache (*cache*).⁴ The *error arrival* activity and the associated distribution determine the probability that an error will impact the cache. A corrupted instruction in the processor cache can be fetched into the execution core (i.e., when the *cache fetch* activity fires, the error token moves from the cache to the execution core) or be evicted from the cache (when the *cache replacement* activity fires, the error token is removed from the cache). These two activities are enabled by the input gate *CPU working* only when the processor is not idle.

An instruction in the execution core can be executed by two mutually exclusive activities: *firewall execution* and *non-firewall workload execution*. The mutual exclusion is guaranteed by two input gates: *firewall enable* and *non-firewall workload enable*; at any time, at most one of the two input gates is enabled. If the executed instruction corresponds to the firewall (*firewall execution* activity fired) four outcomes are possible: (1) no error manifestation, (2) system *crash/hang*, (3) temporary security vulnerability (*T_SV*), or (4) permanent security vulnerability (*P_SV*). The probabilities of the four outcomes are determined from the preceding error injection experiments on the firewall software.

If the executed instruction corresponds to non-firewall workload (*non-firewall workload execution*, activity fired) then either the system crashes or there is no observable impact and the system continues to operate. As mentioned earlier, a temporary security vulnerability disappears when an erroneous instruction is evicted from the processor cache (*cache replacement fired*). To account for this, an extra gate (*rp_out*) is introduced to the SAN model to remove the token from *T_SV* when a corrupted instruction is being removed from the cache.

A *reboot* activity initiated after system crash/hang (or for periodic maintenance) clears all errors. Reboot is the only way of removing a permanent security vulnerability.

8.1. SAN model simulation results

The SAN model was simulated (using Mobius [15]) to obtain measures characterizing the error-caused security vulnerabilities and the resulting security violations. Recall that in our model, at least five malicious packets have to penetrate the firewall for a violation to occur. The reason is that a significant violation can be engineered using as few as five packets.

The measures obtained from the model include: the rate of temporary/permanent security vulnerabilities, the duration of a temporary security vulnerabilities, and the number of potentially malicious packets

⁴ We assume the main memory is protected by error correction code (ECC) and substantially free from errors.

Table 3

Parameter values used in the model

Parameter	Value	Comments
Error rate (<i>error arrival</i>)	0.1 per day	Based on field failure data of operational systems (see [13])
Processor execution rate (<i>firewall execution, non-firewall workload execution</i>)	10^9 s^{-1}	Number of instructions executed per second by 1 GHz processor
Cache miss rate	1%	Cache miss rate of 0.39% was shown for SPEC92 benchmarks executing on DECstation 5000 [18]. In a multi-task environment, it is likely that cache miss rate is higher than 0.39%, because of the factors such as the burst of cache replacement during task switch. Considering these factors, we assume the rather conservative cache miss rate of 1%
Cache fetch rate (<i>cache fetch</i>)	10^5 s^{-1}	A 32 KB on-chip instruction cache can hold approximately 10^4 x86 instructions. Assuming uniform cache access, each cycle, an instruction in the cache has 10^{-4} chance being fetched. For a 1 GHz processor, an instruction can be fetched $10^9 \times 10^4 = 10^5$ times every second
Cache replacement rate (<i>cache replacement</i>)	10^3 s^{-1}	Assuming uniform cache miss with miss rate 1%, an instruction in the cache can be replaced $10^9 \times 10^{-4} \times 1\% = 10^3$ times every second
Processing rate (<i>packet processing, non-firewall workload, idle time</i>)	10^4 s^{-1}	The rate for workload processing and <i>idle time</i> is same as this rate
Job dispatch distribution (job dispatch) ^a		
<i>Firewall utilization</i>	[0.1, 0.8]	
<i>Non-firewall workload</i>	0, 0.1, 0.2	
<i>Idle time</i>	[0, 0.9]	1— <i>firewall utilization</i> — <i>non-firewall workload</i>
Distributions of outcomes from firewall execution (<i>firewall execution</i>); obtained from error injection experiments (refer to Table 2)		
<i>T_SV</i>	2%	Temporary security vulnerability
<i>P_SV</i>	0.05%	Permanent security vulnerability
<i>Crash/Hang</i>	20.22%	System crash
<i>No manifestation</i>	77.73%	Error not manifested
Distribution of the outcomes of the <i>non-firewall workload execution</i>		
<i>Crash/Hang</i>	30%	System crash
<i>No. manifestation</i>	70%	Error not manifested

^aDue to the space limitation, we only present data corresponding to a subset of the combination of firewall utilization and non-firewall utilization.

exploiting the window of security vulnerability and penetrating the firewall. The model was simulated using the parameters specified in Table 3. The table also provides a justification for the choice of parameter values.

We analyze the security implications of error-induced vulnerabilities in the firewall in the context of a large networked system protected by several firewalls, using the simulation results from the SAN model. Consider a networked system in an organization (e.g., a large enterprise or a university campus):

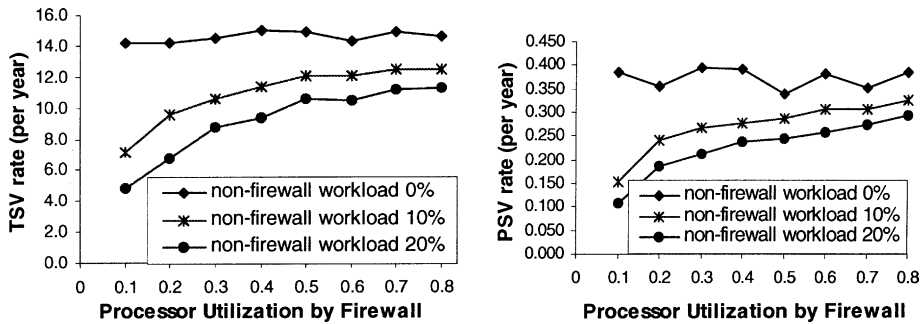


Fig. 8. Rate of temporary security vulnerability (left) and permanent security vulnerability (right) with 0.1 error per day for 20 firewall machines.

the network typically has multiple entry points, each protected by an independent firewall. Error-caused security vulnerabilities, temporary and permanent, could allow several malicious packets to penetrate a firewall. Thus, compromising one of the entry points (i.e., the firewalls) could allow an intruder to potentially access hundreds of machines.

Rates of Temporary and Permanent Security Vulnerabilities. Left and right diagram of Fig. 8 show the rates of temporary security vulnerability (TSV) and permanent security vulnerability (PSV), respectively, as functions of processor utilization by the firewall and the percentage of non-firewall workload. The three curves (in each figure) are derived assuming an error rate of 0.1 error per day and 0, 10, and 20% of non-firewall workload on the processor. Several trends can be observed from the figures:

1. In the absence of the non-firewall workload, the rate of security vulnerability remains relatively unchanged (with the mean values of 14.9 and 0.37 per year for temporary and permanent security vulnerability, respectively, for 20 firewalls) for the entire range of firewall processor utilizations.
2. The presence of non-firewall workload running on the firewall machine reduces the rate of the temporary and permanent security vulnerability. This is because the additional workload increases the chance of an erroneous instruction to be evicted from the cache.
3. For a given firewall processor utilization, the rates of both temporary and permanent security vulnerabilities decrease with increasing non-firewall workload.

A measure of the temporary and permanent vulnerability rates shows how often a window of vulnerability occurs. Clearly, the occurrence of the vulnerability must coincide with the arrival of malicious packets.

Temporary Window of Vulnerability. Fig. 9 shows the mean number of malicious packets that can penetrate the firewall during a temporary window of vulnerability as a function of the firewall processor utilization and the percentage of non-firewall workload. We assume a malicious packet percentage of 30%, which is not unusual; network studies (e.g., [19]) indicate that about 30% of network bandwidth can be taken by attacks and the peak malicious packet percentage can be as high as 60%. Moreover, analysis of data on the IP traffic in our laboratory network indicates that the average percentage of suspected (i.e., potentially malicious) IP flows is as high as 21.3%, with a standard deviation of 6.5%.⁵ The three curves

⁵ The data are collected using IPAudit [27] network monitoring software and the reported figures are calculated based on random sample of 24 half-hour periods from September 2002.

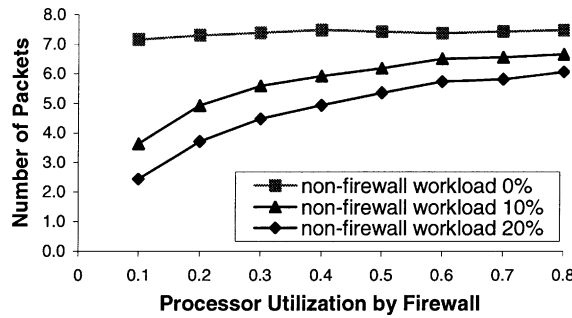


Fig. 9. Mean number of malicious packets penetrating the firewall during a window of temporary vulnerability, 30% malicious packets.

in Fig. 9 are derived assuming an error arrival rate of 0.1 error per day and for 0, 10, and 20% non-firewall workload.

The following can be observed from this figure:

1. In the absence of the non-firewall workload, the mean number of packets that can penetrate during a temporary window of vulnerability remains almost constant (approximately seven packets).
2. Presence of the non-firewall workload running on the firewall machine reduces the mean number of packets penetrating the firewall. For example, for 50% firewall processor utilization, the mean number of malicious packets getting through the firewall drops from around seven for no non-firewall workload, to around six packets for 10% non-firewall workload, and around five for 20% non-firewall workload.

Distribution of the number of malicious packets penetrating during a window of temporary vulnerability. Left and right diagram of Fig. 10 show the distribution of the number of malicious packets penetrating the firewall during a temporary window of vulnerability, assuming that 30% of incoming packets are malicious, 50% firewall processor utilization, and 0% (Fig. 10 (left)) and 10% (right) non-firewall workload. It can be observed that in the absence of the non-firewall workload, the distribution of the window of vulnerability size has a longer tail (about 5% of the windows of vulnerability allow more than 11 malicious packets to get through the firewall). With 10% of non-firewall workload, however, the tail of the distribution is shorter (about 5% of the windows of vulnerability allow more than nine malicious packets to get through the firewall). Approximately 26 and 20% (left and right diagrams, respectively, of Fig. 10)

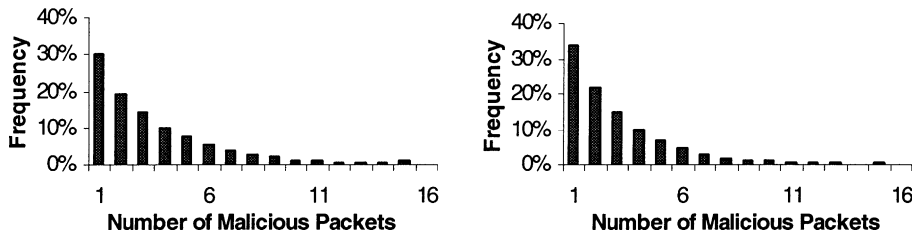


Fig. 10. Distribution: number of malicious packet penetrating during a window of temporary vulnerability, 50% processor utilization by the firewall, 30% malicious packets, 0% (left) and 10% non-firewall workload (right).

Table 4
Rate of kernel-related security vulnerability

Operating system	Number of reported kernel-related security vulnerabilities	Time period	Rate of security vulnerability per year
RedHat Linux (7.0, 7.1, 7.2)	13	November 2000–August 2002	7.4
Solaris 2.6	25	February 2000–August 2002	9.7
Windows 2000	38	February 2000–August 2002	14.7

of the windows of vulnerability allow five or more malicious packets to penetrate the firewall (i.e., cause a security violation). These packets are sufficient to launch a large, real security attack, e.g., a buffer overflow attack, which can lead to severe damage to secure systems.

8.2. Evaluating the frequency of security violations

Two questions to be asked are: How significant are error-caused security vulnerabilities compared to those due to software bugs or system misconfiguration? Given the fraction of malicious packets in the packet stream, how likely is it that a sufficient number of malicious packets (in our case, five or more) will get through the firewall to initiate and execute an actual attack?

Published vulnerabilities in major operating systems. We first examine the reported and publicly available kernel-related vulnerabilities [16]. Using this data, the rate of kernel-related vulnerability in an operational system is estimated. Data on published kernel-related vulnerabilities for three major operating systems—RedHat Linux 7.x, Solaris 2.6, and Windows 2000—is summarized in Table 4.

Error-caused security violations. Tables 5 and 6 show the rates of security violations (i.e., five or more malicious packets penetrating the firewall) for an error arrival rate of 0.1 per day (1 error per 10 days) and for a range of non-firewall workloads, assuming 50 and 80% processor utilization by firewalls, respectively. In our study, we assume that for a typical network, the percentage of malicious packets is in the range of [20%, 40%]. According to Table 5, with a 30% malicious packet percentage, about 2 or 3 security violations occur in a 1-year period for 50% firewall processor utilization and a non-firewall workload of up to 10%. Over a 1-year period, Table 4 shows approximately 7.4 reported vulnerabilities for RedHat Linux.

The reported security vulnerabilities in Table 4 and error-caused security violations in Tables 5 and 6 are caused by different phenomena, the former by software bugs and the latter by transient errors in the

Table 5
Rates of error-caused security violations, firewall processor utilization = 50%

Rate of error-caused security violation (1 per year)	Non-firewall workload = 0% ^a			Non-firewall workload = 10% ^a		
	20%	30%	40%	20%	30%	40%
Due to TSV	1.24	2.65	3.86	0.6	1.54	2.48
Due to PSV	0.34	0.34	0.34	0.28	0.28	0.28
Total	1.58	2.99	4.2	0.88	1.82	2.76

^a20, 30 and 40% are the malicious packet percentage.

Table 6
Rates of error-caused security violations, firewall processor utilization = 80%

Rate of error-caused security violation (1 per year)	Non-firewall workload = 0% ^a			Non-firewall workload = 10% ^a		
	20%	30%	40%	20%	30%	40%
Due to TSV	1.22	2.62	3.84	0.96	1.82	2.84
Due to PSV	0.38	0.38	0.38	0.32	0.32	0.32
Total	1.6	3	4.22	1.29	2.14	3.16

^a20, 30 and 40% are the malicious packet percentage.

system. Once a security-related software bug is discovered, all machines running on the same version of an operating system are susceptible to this bug until the corresponding patch is applied. Error-caused violations, while impacting only a particular machine, can also open access to multiple machines behind the firewall. As the systems are under virtually constant attack, the consequences of such security violations can be severe and unpredictable. For example, the intrusion detection system *SNORT* [28] installed on our network reported 961,786 alerts for the 7-day period from September 23–29, 2002. This gives the mean time to alert of 0.63 s (or the alert arrival rate of 1.59 s^{-1}). These alerts include only activities recognizable by the *SNORT* signature database. A highly secure military network is likely to operate under even harsher conditions and error-caused security vulnerabilities can lead to serious security violations. While the work in this paper shows that security vulnerabilities and security violations can be modeled and strictly defined, it is only a first step in our endeavor toward the goal of quantitative security evaluation.

9. Conclusion

This paper presented an experimental study of error-related security issues in Linux kernel firewalls. The target components of the error injection experiment described here were two widely used firewalls: *IPChains* and *Netfilter*. The error injection experiments revealed that, respectively, 2.05 and 1.95% of the errors injected into the code segment of the two firewalls caused security vulnerabilities. Some of the errors propagated to permanent data and opened long-lasting security holes. Others created temporary vulnerabilities, which disappeared when the errors disappeared. We also proposed a SAN model to describe and analyze the interaction among major factors that impact the probability of a security vulnerability. Results from simulating the SAN model showed that, for example, in a network system protected by 20 firewalls, assuming 0.1 error per day, 50% processor utilization by the firewall, a 10% non-firewall workload, and a 30% malicious packet rate, more than two machines (on average) will suffer from error-caused violations. These violations may cause effective damage to the network, e.g., overflowing buffers on the machines behind the firewall. Thus the error-caused security vulnerabilities can be a non-negligible source of security threat to a highly secure system.

Acknowledgements

This work is supported in part by NSF Grant CCR 00-86096 ITR, 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.

References

- [1] R. Anderson, M.G. Kuhn, Tamper resistance—a cautionary note, in: Proceedings of the Second USENIX Workshop on Electronic Commerce, Oakland, CA, November 1996.
- [2] D. Boneh, R.A. DeMillo, et al., On the importance of eliminating errors in cryptographic computations, in: Advances in Cryptology: Eurocrypt'97, 1997, pp. 37–51.
- [3] A. Ghosh, T. O'Connor, et al., An automated approach for identifying potential vulnerabilities in software, in: IEEE Symposium on Security and Privacy, Oakland, CA, May 1998.
- [4] R.L. Rivest, A. Shamir, L. Adleman, A method for obtaining digital signatures and public key cryptosystem, *Commun. ACM* 21 (2) (1978) 120–126.
- [5] D.T. Stott, B. Floering, et al., Dependability assessment in distributed systems with lightweight fault injectors in NFTAPE, in: Proceedings of the IEEE International Computer Performance and Dependability Symposium, March 2000, pp. 91–100.
- [6] J. Xu, S. Chen, et al., An experimental study of security vulnerabilities caused by errors, in: Proceedings of the International Conference on Dependable Systems and Networks, Göteborg, Sweden, 2001.
- [7] P. Kochar, J. Jaffe, B. Jun, Differential power analysis: leaking secrets, in: Crypto'99.
- [8] R. Russell, Linux IPCHAINS-HOWTO. <http://netfilter.samba.org/ipchains/HOWTO.html>.
- [9] Y. He, A. Avizienis, Assessment of the applicability of COTS microprocessors in high-confidence computing systems: a case study, in: Proceedings of the ICDSN 2000, June 2000.
- [10] M. Rimen, J. Ohlsson, et al., On microprocessor error behavior modeling, in: Proceedings of the IEEE International Symposium on Fault-Tolerant Computing (FTCS-24), June 1994.
- [11] H. Cha, E. Rudnick, J. Patel, et al., A gate-level simulation environment for alpha-particle-induced transient faults, *IEEE Trans. Comput.* 45 (11) (1996) 1248–1256.
- [12] G. Choi, R. Iyer, D. Saab, Fault behavior dictionary for simulation of device-level transients, in: Proceedings of the IEEE International Conference on Computer-Aided Design, 1993.
- [13] I. Lee, R. Iyer, D. Tang, Error/failure analysis using event logs from fault tolerant systems, in: Proceedings of the IEEE International Symposium on Fault-Tolerant Computing (FTCS-21), June 1991, pp. 10–17.
- [14] R. Ortalo, Y. Deswarte, M. Kaaniche, Experimenting with quantitative evaluation tools for monitoring operational security, *IEEE Trans. Software Eng.* 25 (5) (1999) 633–650.
- [15] D. Daly, D.D. Deavours, et al., Mobius: an extensible tool for performance and dependability modeling, in: Proceedings of the 11th International Conference, TOOLS 2000, Schaumburg, IL, March 2000.
- [16] <http://www.securityfocus.com>, December 30, 2001.
- [17] Microsoft Product Security, FrontPage Server Extension Sub-Component Buffer Overflow Vulnerability. <http://www.securiteam.com/windowsntfocus/5YP0MOU4KA.html>.
- [18] D. Patterson, J. Hennessy, *Computer Architecture a Quantitative Approach*, 2nd ed., Morgan Kaufmann, Los Altos, CA, 1996, 384 pp.
- [19] G. Xu, H. Zhang, Advanced methods for detecting unusual behaviors on networks in real-time, in: Proceedings of the IEEE International Conference on Communication Technology, vol. 1, WCC-ICCT 2000, 2000, pp. 291–295.
- [20] M. Joye, J.-J. Quisquater, F. Bao, R.H. Deng, RSA-type signatures in the presence of transient fault, in: *Cryptography and Coding*, Lecture Notes in Computer Science, vol. 1335, Springer, Berlin, 1997, pp. 155–160.
- [21] A. Menezes, P. van Oorschot, S. Vanstone, *Handbook of Applied Cryptography*, CRC Press, Boca Raton, 1996, pp. 284–291.
- [22] D. Stott, Automated fault-injection-based dependability analysis of distributed computer systems, Ph.D. Dissertation, University of Illinois at Urbana-Champaign, Urbana, IL, 2001.
- [23] D. Wreski, Linux Kernel 2.4 Firewalling Matures: Netfilter. http://www.linuxsecurity.com/feature_stories/kernel-netfilter.html.
- [24] R. Ronen, A. Mendelson, K. Lai, et al., Coming challenges in microarchitecture and architecture, *Proc. IEEE* 89 (3) (2001) 325–340.
- [25] J.F. Ziegler, et al., IBM experiments in soft fails in computer electronics (1978–1994), *IBM J. R&D* 40 (1) (1996) 3–18.
- [26] PCI PC Watchdog Board User's Manual, Berkshire Products Inc. http://www.berkprod.com/pci_pc_watchdog.htm.
- [27] IPAudit. <http://ipaudit.sourceforge.net/>.
- [28] SNORT: The Open Source Network Intrusion Detection System. <http://www.snort.org/>.