

A Study of the Error Impact on System Security

Jun Xu, Shuo Chen, Zbigniew Kalbarczyk, Ravishankar K. Iyer

Center for Reliable and High-Performance Computing

University of Illinois at Urbana-Champaign

1308 W. Main Street, Urbana, IL 61801

Phone: 217-244-6104, Fax: 217-244-5686

{junxu, shuochen, kalbar, iyer}@crhc.uiuc.edu

Abstract

Designers of large networked systems are becoming more concerned of about both fault tolerance and system security. To build such systems, it is critical to understand the relationship between errors and security vulnerabilities. In this paper, we test the hypothesis that errors, in particular control flow errors, can cause security vulnerabilities. We identify the user authentication sections of ftpd (file transfer protocol server) and sshd (secure shell server) applications and conduct error injections to study the sensitivity of system integrity to errors. Results show that, out of all activated errors: (1) 1% to 2% of errors compromise system security (create a permanent window of vulnerability), (2) 43% to 62% of errors result in crash failures (about 8.5% of these errors create a transient window of vulnerability), and (3) 7% to 12% of errors result in fail silence violations. We analyze reasons why a single bit error to create a security hole or crash the system. Finally, we present the design and evaluation of a new encoding scheme for branch instructions that reduces or eliminates cases in which a single bit error compromises the system integrity.

Keywords: Control-flow errors, security vulnerability, experimental assessment, instruction set encoding, error injection

Submission Category: Regular Paper

1. Introduction

Networked systems, such as large web server farms and e-commerce transaction systems, are often running under high load, faults, errors, and security attacks. Designers of such systems frequently seek to implement fault tolerance as well as a high level of security. However, these two goals impose conflicting requirements on system design and implementation. On the one hand, fault tolerance requires redundancy, such as replication and distributed processing; on the other hand, security requires a single point of control and centralized processing. When designing such systems, it is critical to understand the relationship and interaction between errors and security problems.

In this paper, we study the impact of errors on system security. The goal of this work is to study the hypothesis that errors, in particular control flow errors¹, can compromise system security. We test this hypothesis in the context of two widely used Internet server applications, *ftpd* (FTP daemon, server program for Internet File Transfer Protocol) and *sshd* (SSH daemon, server program for Secure Shell Protocol). Both programs require user authentication before granting access to server resources. Analyzing the user authentication sections of the source code of these two applications, we determine that due to the structure of the code, corruption to the control flow instructions can potentially subvert the programmer's original intended flow of control and open the system to intruders. We use error injection to create realistic failure scenarios and to analyze their impact on system security. In particular, we seek to determine whether corruption to control flow instructions (even by a single bit flip) can make the system vulnerable to security attacks. We focus on control errors, as they can lead to data corruptions (e.g., in the database), process crashes, fail silence violations, and, more importantly, security vulnerabilities. Another important reason that we focus on control flow errors is that the applications under study, in particular their authentication sections, are control-intensive and therefore, more sensitive to control flow errors. To the best of our knowledge, this is the first study on the impact of control flow errors on system security.

The key contributions of this paper can be summarized as follows:

1. Analysis of the impact of control flow errors on system security;
2. Identification of two types of system vulnerability windows: (1) permanent and (2) transient. A *permanent vulnerability window* is the time period within which the system (due to an error) becomes permanently

¹Errors seen by an application can be broadly classified as data errors and control errors. *Data errors* affect the values of variables, or registers of the application. *Control errors* change the control flow of the application and cause a divergence from the intended execution path.

open to an intruder (i.e., until the application is reloaded, swapped, or the system is rebooted). A *transient vulnerability window* is the time period between activation of an error (i.e., execution of an erroneous instruction) and the occurrence of application/system crash. We show that a *transient* window can include the execution of more than 16,000 instructions (not counting those executed inside the kernel). During this time, the error can propagate, and as a result, the application may send erroneous messages to other participants in the network or wrongfully change its internal state, or it may compromise other system components;

3. Analysis of key reasons why a single bit error can create a security hole or crash the system. Design and evaluation of a new encoding scheme for branch instructions (on Intel x86) to reduce or eliminate cases in which a single bit error compromises the system integrity.

Results show that out of all activated errors: (1) 1% to 2% of errors compromise system security (create a *permanent window of vulnerability*), (2) 43% to 62% of errors result in crash failures (about 8.5% of these errors create a *transient window of vulnerability*), and (3) 7% to 12% of errors result in fail silence violations. Detailed analysis shows that most of the security break-ins and many of the fail silence violation cases are caused by the program taking a valid but incorrect path in the presence of errors. A detailed breakdown of these cases shows that a single bit error can change a branch instruction opcode to another valid (but incorrect) opcode and therefore subvert the originally intended execution flow. The reason for such a radical change under the single bit error model is that in the Intel x86 architecture [5] instructions are encoded so that the minimum Hamming distance between individual conditional branch instructions is equal to one. We propose and evaluate a new encoding scheme to increase the Hamming distance between the conditional branch instructions so that single bit errors will not lead to illegal changes in the control flow of the application.

2. Related Work

In recent years, system security has been an area of intense research. The rapid growth of World Wide Web and, in particular, the development of e-commerce make security a critical issue in providing network services. Nevertheless, publicly available information on robust solutions in building secure systems is rather limited compared with the volume of studies and information on designing highly fault-tolerant system. Most previous work on security focuses on authentication protocols [12, 19], encryption [16], intrusion detection, and anomaly detection [2]. More recently, researchers from LAAS of France and IBM began studies on quantitative measurement of operational security [4, 13]. Also emerging are studies on the impact of environmental factors in intrusion detection

systems [10]. Our work brings together the study on errors and system security. To best of our knowledge, this is the first study that considers impact of errors (in the text segment of an application) on system security.

Errors in the application control flow have been demonstrated to cause severe consequences, including system crashes and fail-silence violations (which lead to error propagation). The impact and protection against control flow errors have been studied for quite some time. Mahmood[9] presents a survey of techniques in hardware for detecting control flow errors. Recent years have brought several software-based techniques (implemented either at the assembly level or at high-level language level). The basic scheme is to divide the application program into basic blocks. Examples include Block Signature Self Checking[11], Enhanced Control Checking with Assertions[1] and PECOS[3]. While these techniques detect invalid flow of controls in a program, the proposed instruction set encoding scheme detects valid but incorrect conditional branches due to a change of opcodes. The encoding scheme presented in this paper is complimentary to these related control flow error detection techniques.

3. Impact of Errors on System Security: Examples

In this section, first we briefly introduce the two target applications and then discuss examples from them illustrating how data and control errors can create security holes.

3.1. Target Applications

File Transfer Protocol (FTP) [15], is an Internet protocol used for transferring files between a file server and a client host. A user logon to an FTP server, authenticates itself using user name and password, and retrieves or uploads files from/to the file server. We use *wu-ftp-2.6.0*, a widely used server implementation from Washington University. The user authentication part of this implementation includes two functions, `user()` and `pass()`, which check the user identity and password and award access if both checks pass. These two functions have 1211 lines of C source code, constituting about 5.8% of the entire *wu-ftp-2.6.0* source base and about 8% of the entire compiled binary code. The target for error injection, i.e., branch instructions, in these two functions accounts for 7432 bits (about 13% of these two functions).

Secure Shell (SSH) [19], is a program used to log into another computer over a network, execute commands in a remote machine, and move files from one machine to another. SSH provides strong authentication and secure communications over insecure channels. SSH is most useful for logging into a UNIX computer from a remote machine in cases in which the traditional *telnet* [14] and *rlogin* [8] programs would not provide password and session

encryption. We use the *ssh-1.2.30* distribution by *SSH Communications Security*, Espoo, Finland. The user authentication part of this implementation which we study includes the three functions `do_authentication()`, `auth_rhosts()`, and `auth_password()`. These three functions have 1236 lines of C source code, about 3.6% of the entire SSH source code base, and 2.1% of the entire compiled binary code. The target for error injection, i.e., branch instructions, in these three functions accounts for 2664 bits (about 12% of the three functions).

3.2. Example 1

The first example is taken from function `pass()` of *wu-ftpd-2.6.0*. This function is responsible for checking whether the remote user's password is correct and granting/denying access to the server accordingly. In this example, the C code checks whether the provided password matches the system stored password and sets a grant flag if they match. The C source code as well as disassembled binary code are shown in Figure 1.

C Source Code

```
if ( ... && (strcmp(xpasswd, pw->pw_passwd) == 0) ) {
    rval = 0;
}
if ( rval ) {
    /* deny access */
    ...
}
/* grant access */
...
```

Disassembled Binary Code

```
<pass+216>:  push    %eax           # pw->pw_passwd
<pass+217>:  push    %ecx           # xpasswd
<pass+218>:  call   <strcmp>       # call strcmp(...)
<pass+223>:  add     $0x8,%esp     # shrink stack
<pass+226>:  test   %eax,%eax     # test if return value is 0
<pass+228>:  jne    <pass+232>     # if not 0, jmp to <pass+232>
<pass+230>:  xor    %ebx,%ebx     # if 0, rval=0
<pass+232>:  test   %ebx,%ebx     # test if rval is 0
<pass+234>:  je     <pass+1203>    # if 0, jump to grant part
<pass+240>:  push   $0x8062907    # if not 0, deny
                ... deny access and return ...
<pass+1203>:  ... grant access ...
```

Figure 1. Example from function `pass()` of *ftpd*

In the code segment shown in Figure 1, we identify three instructions that can subvert the integrity of the server with a single bit error. One instruction provides a function argument, and the other two are conditional branch instructions that are decision-making points in the server process.

1. At address `<pass+216>`, a single bit flip can change `push %eax` (encoding `0x50` to `push %ecx` (encoding `0x51`), which provides `strcmp()` with two identical strings and makes `strcmp()` always return 0;
2. At address `<pass+228>`, a single bit flip can change `jne` (encoding `0x75`) to `je` (encoding `0x74`) and reverse the branch direction. Instead of branching to `<pass+232>`, the program fall through to `<pass+230>` and sets `%ebx (rval)` to 0;
3. At address `<pass+234>`, it behaves similarly, except it may change `je` to `jne`. Instead of falling through to the deny part in case of a wrong password, it branches to the password accept part;

In all three cases, the server will grant access to the system for anyone who logs in with an existing user name (relative easy to obtain) and an arbitrary or invalid password. Observe that this situation creates a permanent security hole which can be eliminated only through application reload or system reboot. The reason this can happen is that the Hamming distance between the opcodes of the two instructions equals to one. Therefore, a single bit error can change `push %eax` to `push %ecx` and `je` to `jne` or vice versa. Such a change completely reverses the control flow of the program and results in a security breach.

3.3. Example 2

This example is taken from function `do_authentication()` of `sshd`. This function uses a combination of mechanisms to authenticate a remote user. The code segment shown in Figure 2 is one of the authentication mechanisms. namely `auth_hosts(...)`, which returns a non-zero value when the remote user is awarded access to the system. Again, if the `je` instruction in the disassembled code is changed to `jne`, the flow of control is subverted and results in a security violation, i.e., an unauthorized user can get into the system.

3.4. Example 3

This example is taken from function `packet_read()` of `sshd`. It shows how a data error can affect the secure operation of the server. When function `read(...)` is called to receive a packet from the network, it performs

C Source Code

```
if (auth_rhosts( ... ))
{
    /* Authentication accepted. */
    ...
    authenticated = 1;
    break;
}
```

Disassembled Binary Code

```
call    0x804df20 <auth_rhosts>    # call auth_hosts
mov     %eax,%edx                 # %edx = return value
add     $0x18,%esp
test    %edx,%edx                # if %edx == 0
je      0x804bda6 <%eip+33>       # yes, deny
...
...                               # no, accept
movl   $0x1,0xffffffff(%ebp)     # authenticated = 1;
...
jmp     0x804c093                 # break to accept
```

Figure 2. Example from function `do_authentication()` of `sshd`

buffer overflow checking, i.e., it verifies whether the size of the incoming data does not exceed the predefined data buffer, `buf`. An error that alters the immediate number `0x2000` (decimal 8192) in the push instruction or a data error on calling stack for `read()`, or a control flow error inside the `read()` function when the buffer boundary checking is executed can create an opportunity for stack overflow attacks, i.e., hijack the server process.

The examples presented in this section provide convincing evidence that errors in the code segment of an application can make system vulnerable from the security perspective. More importantly, any of the discussed scenarios creates permanent security hole. These holes do not crash the system, but as long as the system is not rebooted or the memory pages are not reloaded, any user can log in without proper authentication.

4. Experimental Approach

To better understand how errors impact the security of the target applications and to assess the likelihood of compromising system security due to errors in the text segment, we conduct a set of error-injection-based experiments. We use *Selective Exhaust Injection* as a trade-off between random and exhaustive error injections. Random injection is an effective way to characterize failure behaviors when the target application is large, but it

C Source Code

```
int packet_read(void)
{
    char buf[8192];

    ...
    /* read packet from network */
    len = read(connection_in, buf, sizeof(buf));
    ...
}
```

Disassembled Binary Code

```
push    $0x2000                # sizeof(buf)
lea     0xffffdf80(%ebp),%esi  # buf
push    %esi
pushl   0x8077604              # connection_in
call    804a4a8 <read>        # call read
```

Figure 3. Example from function `packet_read()` of `sshd`

suffers from an inability to fully understand the impact and distribution of errors. Exhaustive injection, on the other hand, provides a complete view of error impact but suffers from the prohibitive cost of time and computational resources. Our method is a trade-off between the two. It is *selective* in the sense that we choose only the code segments most relevant to our evaluation goal, i.e., the authentication sections of the executables, and we conduct error injections in the selected regions of the code segment. These parts are critical for the integrity of the systems from a security point of view. The error injections are *exhaustive* in that we run experiments until every bit of every branch instructions in the selected segments is injected. For each experiment, we flip one bit in the code and run the server until completion of one client connection. For example, instruction

```
74 05                je     $PC+5
```

has two bytes (16 bits). We run 16 experiments for this instruction each running with one of the 16 bits corrupted. Injecting one error at a time allows us to observe the exact consequence of each error. If multiple bit errors are used in each experiment, it is often hard to trace exactly which of them resulted in a crash or semantic violation.

Error injection campaigns are performed using NFTAPE [17], a comprehensive set of tools for fault and error injections in networked environments. A debugger-based injector from NFTAPE is used. The injector loads the server executable into memory, sets a breakpoint at the instruction where an error is to be injected, and starts

running the server process. In the meantime, a client for the server is started on another machine and tries to log onto the server. If the chosen instruction is on a path that the server needs to execute for a specific run, the server stops at the pre-set breakpoint and the injector injects the error at that time and continues execution of the server process. However, if the chosen instruction is not on an execution path, the server runs to completion without the breakpoint being activated. The above mentioned procedure is used to monitor whether the corrupted instruction is actually executed. If the error is activated and causes the server process to crash, the injector intercepts the signal and logs it before the server process is terminated. Otherwise, output from the server process and the injector are logged for off-line analysis.

5. Experimental Results and Analysis

This section presents the results from error injection campaigns on *sshd* and *ftpd* and discussions of our findings.

5.1. Results Categorization

In the conducted error injection experiments, the activation and execution of an erroneous instruction leads to different types of outcomes. We categorize outcomes into the following five types:

Not Activated (NA). The breakpoint is not reached during the execution and therefore, the client and server operate in a normal manner.

Activated but Not Manifested (NM). The breakpoint is reached and the corrupted instruction is executed, but the error has no impact on the server and the client gets the requested service.

System Section (SD). The corrupted instruction is executed and the server crashes because of the error. The crash is usually caused by an illegal instruction or segmentation violation.

Fail Silence Violation (FSV). The corrupted instruction is executed but the communication pattern and/or data exchanged between the server and client is not consistent as observed in an error free execution. That means the error causes the server to take a different execution path and results in violation of the intended flow of control. In the context of our selected applications, examples of FSV are skipping sending a message, sending an extra message, denying access to a resource when it should act otherwise.

Security Break-in (BRK). This is a special type of FSV which creates security holes. The manifestation of BRK is that the server program awards access to the client when it should not do so. In *ftpd*, a break-in

means a client successfully logged in and retrieved files from the server; In *sshd*, it means the remote client successfully gets a login shell when it should not.

5.2. FTP Error Injection Results and Analysis

Errors are injected into the branch instructions in the two selected functions `user()` and `pass()`. Four different client access patterns are used to log on to the server and to characterize the server behavior. *Client1* uses an existing user name but a wrong password, emulating a security attack from an unauthorized client. *Client2* uses an existing user name and a correct password. *Client3* uses a non-existing user name and password. *Client4* logs on as an anonymous user. All clients try to retrieve several files if the server authorized the login.

Type	Client1		Client2		Client3		Client4	
NA	6776	-	6384	-	6936	-	6176	-
NM	307	46.80%	410	39.12%	190	38.31%	378	30.10%
SD	285	43.45%	517	49.33%	273	55.04%	785	62.50%
FSV	57	8.69%	121	11.55%	33	6.65%	93	7.40%
BRK	7	1.07%	-	-	-	-	-	-

Table 1. FTP Result Distributions

Table 1 shows the results distribution for 7432 runs, corresponding to the number of bits in all branch instructions in the target functions. There are two columns for each client, the left column shows the raw number for each result category, and the right column shows the percentage against all *activated* errors. A dash (-) denotes not applicable. This format is used throughout this paper. On average, about 88% of all errors are never activated. The low activation rate is because large chunks of the selected code are not executed during a run of one particular type of client request. For example, when *Client1* attempts to log on as a normal user, the code that handles anonymous login is not reached. *Client1* uses a wrong password, thus the code segment that handles correct passwords and therefore grants access is not reached. The four selected client access patterns exercised most of the two target functions.

About 38.5% of activated errors have no impact on the correct execution of the server and client. Such non-manifestation is because the injected bit errors do not change the type of an opcode. About 52% of all activated errors cause a system detection and the server process to crash. These errors make an instruction invalid, change the offset of the branch instruction to a invalid location or to a location in the middle of an valid instruction, or they change a branch instruction to another type of instruction and cause the register or memory state to be

changed. About 9% of all activated errors caused fail silence violations. We observe different manifestation of FSV. Sometimes the server process sends an extra or a wrong message that confuses the client; sometimes the server skips sending a required message that the client is awaiting, making the client to hang; sometimes the server grants/denies access for a client while the protocol indicates it should act otherwise. As an example, consider one of the runs for *Client1*. The client sends user name to the server and expects an acknowledgement. Due to a control flow error in the server process the server, instead of sending an acknowledgement to the client, branches to an erroneous location and sends an invalid reply message that *confuses* the client. Note that the server process ultimately crashes due to internal state problem caused by the error. Of particular interest for our study, there are 7 cases of BRK compromise the security of the server in *Client1*. In these cases, the client obtains access to the system despite an invalid password.

5.3. SSH Error Injection Results and Analysis

As in *ftpd*, errors are injected into branch instructions in the selected three user authentication functions, `do_authentication()`, `auth_rhosts()` and `auth_password()`. We apply two client access patterns. *Client1* logs on to the server using an existing user name but a wrong password; *Client2* used an existing user name and a correct password. Table 2 shows the results distribution for all 2664 runs, corresponding to the number of

Type	Client1		Client2	
NA	1424	-	1408	-
NM	498	40.16%	500	39.81%
SD	650	52.42%	659	52.47%
FSV	73	5.89%	97	7.72%
BRK	19	1.53%	-	-

Table 2. SSH Results Distributions

bits in all branch instructions in the target functions. The percentage columns in the table are computed against all *activated* errors. On the average, about 40% of all errors are never activated. Compared to the results from *ftpd*, *sshd* has much higher error activation rate because the C source code in the *sshd* implementation is more compact than that of *ftpd*. About 40% of activated errors have no impact on the correct execution of the server and client. About 52% of activated errors cause a system detection and the server process crashes. About 7.5% of activated errors caused fail silence violations. The manifestations of fail silence violations are similar to those observed from *ftpd*.

About 1.5% (19 cases) of activated errors from *Client1* open the system to security attacks. Comparing the percentages of BRK from *ftpd* and *sshd*, we observe that *sshd* has a higher break-in rate than *ftpd* (1.1%, 7 cases). Analysis of protocol and source code reveals the reason for this difference. In *ftpd*, user name and password checking is the only mechanism of authentication, i.e., there is only a single point of entry to the system. In *sshd*, combinations of mechanisms such as RSA[16](public key authentication), UNIX password, and rhosts (similar to *rlogin*[8]) can be used for user authentication. For each client there exist multiple points of entry into the system. Errors in any of these checks can compromise the integrity of the system. From a security point of view, a single point of control is always preferred. Given that an error changes the control flow of an application, applications with multiple points of entry have a higher probability of being compromised than those with a single point of entry.

5.4. Discussion

Persistent and Latent Errors. From the results presented above, we see that when a error manifests, it either results in fail silence violation or a system detection. Because most errors are not activated or not manifested, one can argue that the chance of an error causing any problem is very small. However, when an error occurs in the system, either in physical memory or stable storage, it persists until the memory page is reloaded or the system is rebooted. Consequently, we have a permanent condition that keeps crashing the server or causing security vulnerabilities and fail silence violations. Client requests with similar access patterns will cause the server to fail in similar fashions.

Impact of System Load. Previous work [6, 7, 18] shows that a program under heavy load tends to have more error manifestations than one under a light load. This is true for our experiments also. The server programs under our study use the following processing model: (1) the main server process listens on the server port for client connections; (2) upon an incoming client request, the main server process forks off a child process to handle the client request. In this processing paradigm, errors stay in memory and remain latent for all subsequent client handling processes. A higher server load means more client requests coming in and the potential for more diversified client request patterns. The more diversified client requests are, the higher chance of different parts of the server code being exercised and thus the higher the probability of a latent error being manifested.

Transient window of vulnerability. We studied the crash failures in detail. In particular, we examine the manner in which errors cause crash failures and what the server is doing between execution of an erroneous instruction

and crash. Figure 4 shows the distribution of the number of machine instructions executed for the crashing process between the error activation point and the crash point from FTP *Client1* experiment. These numbers do not include the instructions executed inside the kernel. Note that X axis is in log scale.

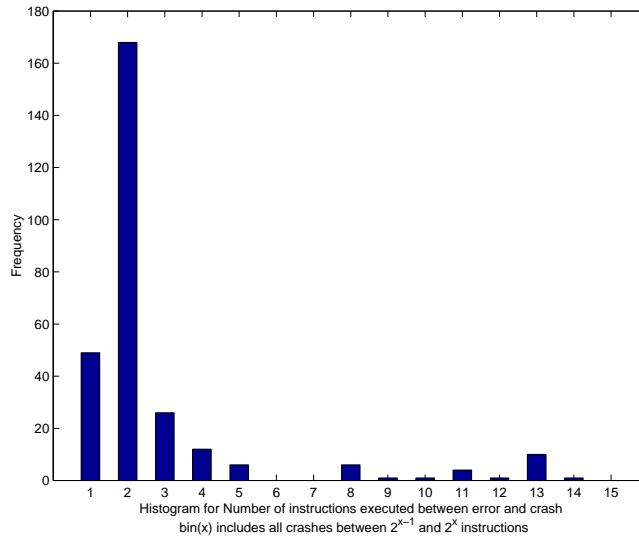


Figure 4. Number of Instructions Executed between Error and Crash. X is in log scale.

Majority (91.5%) of crash failure cases occur within less than 100 instructions after a corrupted instruction is executed. However, in the remaining 8.5% cases, the server executes hundreds, thousands, or even tens of thousands of instructions before it crashes, not counting the instructions executed inside the kernel due to system calls. To the best of our knowledge, this is the first study that tries to provide insight into how a system behaves between error activation and system crash. Those crash failures with long latency are extremely dangerous with respect to security, as they create a transient window of vulnerability to the outside world. The server processes can potentially send out erroneous messages to the clients or incorrectly process messages received from clients. Closely examining some of the crash cases with long latency, we find that several of them send out erroneous messages to the client or receive messages from client (refer to an example discussed in Section 5.2). Although in the limited cases examined we do not observe any security hole open caused by the error, we argue that the potential for such an error causing security problems is not negligible.

6. New Instruction Set Encoding Scheme

In this section, we analyze why corrupting control flow instructions with a single bit flip error can cause failures such as security break-ins and crash failures. We also propose a solution to eliminate these error types.

Examining the BRK and FSV cases observed in the error injection campaigns, we find that failures have two causes: (1) change in a branch instruction’s offset and (2) change in a branch instruction’s opcode. In the case of offset change, the program branches to a different location other than the intended one and either executes some extra code or skips execution of code that would be executed in an error free case. In the case of opcode change, (e.g., a `je` (jump if equal) instruction is changed `jne` (jump if not equal)), the program takes a valid but incorrect branch. Table 3 shows the locations inside an instruction where the errors are injected.

Abbreviation	Definition
2BC	Opcode of 2-byte conditional branch instruction
2BO	Operand of 2-byte conditional branch instruction
6BC1	Byte 1 of Opcode of 6-byte conditional branch instruction
6BC2	Byte 2 of Opcode of 6-byte conditional branch instruction
6BO	Operand of 6-byte conditional branch instruction
MISC	Others

Table 3. Error Location Abbreviations

Tables 4 and 5 show the breakdown of error injection results according to definitions in Table 3. From these two tables, we observe that between 38% and 63% of the BRK and FSV cases are caused by a single bit error in the opcode of a 2-byte conditional branch instructions. About 6.5% to 18% of the cases are caused by an error in the second opcode byte of a 6-byte conditional branch instruction. A closer examination reveals that a vast majority occur because a single bit error causes a conditional branch instruction to change to another conditional branch instruction and therefore subverts the intended path of execution.

Location	Client1		Client2		Client3		Client4	
2BC	39	60.94%	76	62.81%	20	60.61%	46	49.46%
2BO	16	25.00%	23	19.01%	7	21.21%	11	11.83%
6BC1	2	3.12%	4	3.31%	2	6.06%	5	5.38%
6BC2	7	10.94%	14	11.57%	3	9.09%	17	18.28%
6BO	0	0.00%	2	1.65%	1	3.03%	10	10.75%
MISC	0	0.00%	2	1.65%	0	0.00%	4	4.30%
Total	64	-	121	-	33	-	93	-

Table 4. FTP Breakins and Fail Silence Violations Breakdown

The reason for such a critical change under the single bit error model is that the Intel x86 instruction set [5] currently uses continuous encoding of all the conditional branch instructions (also observed in the Sun SPARC

Location	Client1		Client2	
2BC	41	44.57%	37	38.14%
2BO	22	23.91%	26	26.80%
6BC1	0	0.00%	0	0.00%
6BC2	6	6.52%	7	7.22%
6BO	7	7.61%	8	8.25%
MISC	16	17.39%	19	19.59%
Total	92	-	97	-

Table 5. SSH Breakins and Fail Silence Violations Breakdown

instruction set). On an x86, there are two sets of conditional branch instructions, 2-byte and 6-byte². The 2-byte set has one byte of opcode and one byte of branch offset; the 6-byte set has two bytes of opcode and 4 bytes of branch offset; Opcodes of both sets are continuously encoded. The opcodes for the 2-byte set ranges from 0x70 to 0x7F, and the opcodes of the 6-byte set ranges from 0x0F80 to 0x0F8F. Continuous encoding makes processor implementation much easier due to fast instruction decoding, microcode lookup, and executions. This however, means that the minimum Hamming distance between the opcodes of instructions in the same set is one. As a result, single bit error can change one conditional branch instruction to another and thus change the control flow intended by the programmer.

6.1. New Encoding Scheme

We propose a new instruction set encoding scheme that increases the Hamming distance between the block of conditional branch instructions and eliminates the possibility of a single bit error subverting the flow of control.

In the current x86 instruction set, opcodes of 2-byte conditional branch instructions are encoded from 0x70 to 0x7F and opcodes of 6-byte branch instructions ranges from 0xF080 to 0xF08F. The minimum Hamming distance in these two sets is one, therefore a single bit flip can change one conditional branch instruction to another valid conditional branch instruction, reversing the intended flow of execution. For example, in the 2-byte case, `je` is 0x74 and `jne` is 0x75, the Hamming distance between the two is one. A single bit flip can change `je` to `jne` or vice versa. The implication is that a denial of access to a resource on the system becomes grant of access and thus compromises the integrity of the system.

To solve this problem, we shall increase the minimum hamming distance between opcodes of any 2-byte or 6-byte conditional branches so that a single bit flip for any of these instruction will not result in another conditional

²We do not consider the 16-bit branch target offset here because all our experiments are conducted on Linux in 32-bit addressing mode.

branch. In the proposed solution, we increase the minimum Hamming distance between the instructions to at least two. We achieve this through re-shuffling the encoding of the current instruction set. Table 6 shows the mapping from encoding in the old instruction set to encoding in the new instruction set. To achieve a minimum Hamming distance of two, we use the last bit of the most significant four bits of the old opcode as the parity bit for the least four significant bits (we use odd parity). Note that any parity encoding has a minimum hamming distance of two. For example, `jo` with encoding `0x70` has a binary representation of `0111 0000`, the odd parity bit for the lower four bits `0000` is 1 (which we already have), therefore the encoding of `jo` in the new instruction set remains `0111 0000`. On the other hand, `jno` with encoding `0x71` has a binary representation of `0111 0001`, the odd parity bit for the lower four bits `0001` is 0, therefore we need to change the last bit of the higher four bits to 0 and the resulting new encoding is `0110 0001` (`0x61`). Doing so, the new encoding for some of the branch instructions uses the encoding of non-branch instructions in the original encoding scheme. To eliminate this conflict, we swap the encoding of the non-branch instruction with the branch instruction, e.g., in the case of `jno`, we used `0x61` for it in the new encoding and use `0x71` for `popa`, which has an encoding of `0x61` in the old instruction set. The mapping for the opcode of 6-byte instructions is done similarly to the second byte of their opcodes.

Table 6 shows how conditional branch opcodes from the old instruction set are mapped into the new instruction encoding scheme. Columns *2-byte Old* and *6-byte Old* are mapped to columns *2-byte New* and *6-byte New*, respectively. Table 6 shows only the conditional branch mappings. It does not include the swapped non-branch instructions. In the next subsections, we present our evaluation approach and experimental results from the new encoding scheme.

6.2. Experimental Approach

To evaluate the new encoding scheme, we need (1) to build a new processor to incorporate the proposed encoding or (2) to implement the encoding in a processor simulator. A real processor is the preferred option, but it is not possible to change the current x86 processor and build a new one. Simulation cannot emulate the real machine environment needed to conduct error injection experiments and obtain realistic results. We propose a novel approach to testing the new encoding scheme on an existing x86 processor under error injections.

We assume the existence of a hypothetical processor that incorporates the new instruction encoding. Whenever we pick an instruction from the text segment for error injection, we map this instruction from the old encoding to the new one. We then pick a bit in the mapped new instruction to obtain an erroneous instruction in the new

Mnemonics	2-byte Old	2-byte New	6-byte Old	6-byte New
JO	70	70	0F 80	0F 90
JNO	71	61	0F 81	0F 81
JB	72	62	0F 82	0F 82
JNB	73	73	0F 83	0F 93
JE	74	64	0F 84	0F 84
JNE	75	75	0F 85	0F 95
JNA	76	76	0F 86	0F 96
JA	77	67	0F 87	0F 87
JS	78	68	0F 88	0F 88
JNS	79	79	0F 89	0F 99
JP	7A	7A	0F 8A	0F 9A
JNP	7B	6B	0F 8B	0F 8B
JL	7C	7C	0F 8C	0F 9C
JNL	7D	6D	0F 8D	0F 8D
JNG	7E	6E	0F 8E	0F 8E
JG	7F	7F	0F 8F	0F 9F

Table 6. x86 Conditional Branch Instruction Encoding Mapping

encoding. The erroneous instruction is then mapped back to the old instruction encoding and is executed on the processor. One can easily conceive that this process can accurately emulate error injection for the new encoding in the old processor. The mapping can be easily derived from Table 6.

As an example, consider instruction `je` (0x74, 01110100 binary) from the text segment of a current x86 executable. This instruction is mapped to the new encoding scheme using Table 6, we get 0x64, binary 01100100. Assume that the least significant bit is flipped (from 0 to 1); as a result, we get 0x65 or binary 01100101. This value is mapped back to the old instruction encoding and results in 0x65. Note that any encoding now shown in Table 6 remains the same in both old and new encodings. Yet another example, if 0x65 in the old instruction set is to be injected, we map it to the new encoding, which is still 0x65, flip the least significant bit to obtain 0x64. This is then mapped back to the old encoding giving us 0x74 (`je`). This `je` is then executed on the current processor.

6.3. Experimental Results and Discussion

The error injection campaigns described in Section 5 are repeated under the new instruction encoding scheme. Tables 7 and 8 show the results from the new experiments for `ftpd` and `sshd`. Comparing Tables 1 and 2 with Tables 7 and 8, we observe significant reduction in BRK and FSV using new instruction encoding scheme. The last two

rows in Tables 7 and 8 show the BRK and FSV reduction percentage. In case of BRK which affects the system security, the reduction is 86% for *ftpd* and 21% for *sshd*. In case of FSV which are the none security related fail silence violations, the reduction is 21% to 40% for *ftpd* and 34% to 38% for *sshd*.

Type	Client1		Client2		Client3		Client4	
NA	6776	-	6384	-	6934	-	6175	-
NM	234	35.67%	306	29.20%	150	30.24%	284	22.61%
SD	381	58.08%	670	63.93%	320	64.52%	907	72.21%
FSV	40	6.10%	72	6.87%	26	5.24%	65	5.18%
BRK	1	0.15%	-	-	-	-	-	-
FSV Reduction	17	30%	49	40%	7	21%	28	30%
BRK Reduction	6	86%	-	-	-	-	-	-

Table 7. FTP Result from New Encoding

Type	Client1		Client2	
NA	1424	-	1408	-
NM	343	27.66%	342	27.23%
SD	837	67.50%	850	67.68%
FSV	45	3.63%	64	5.10%
BRK	15	1.21%	-	-
FSV Reduction	28	38.36%	33	34.02%
BRK Reduction	4	21.05%	-	-

Table 8. SSH Results from New Encoding

Breakdown analysis similar to those presented in Tables 4 and 5 show that the reductions shown in Tables 7 and 8 are due to the new encoding scheme. In particular, BRK and FSV reductions due to opcode of 2-byte conditional branch and second opcode byte of 6-byte conditional branch instructions account for all the reductions in Tables 7 and 8.

7. Conclusion

In this paper we show that naturally occurring errors in the text segment of an application can cause security vulnerabilities. We demonstrate this through error injection experiments conducted on two Internet applications, *ftpd* and *sshd*. The results show that, given that an error hits the selected program segment, there is a high probability that it will create a security vulnerability, fail silence violation, or system crash. The analysis reveals that security and fail silence violations are caused mostly by the program taking a valid but incorrect branch due

to a single bit error. We present the design and evaluation of a new encoding scheme for branch instructions that reduces or eliminates cases in which a single bit error compromises system integrity.

Closer analysis of crash failures reveals that, although in most cases the system crashes immediately, there are cases in which the process executes thousands or even tens of thousands of instructions before crash. During this period, the process can send erroneous messages or erroneously change the program's internal data structures and potentially compromise system security. This work provides a foundation for further studies (using new applications, such as Web servers) of the relationship between errors and security vulnerabilities.

References

- [1] Z. Alkhalifa, V. Nair, N. Krishnamurthy, and J. Abraham. Design and Evaluation of System-Level Checks for on-line Control Flow Error Detection. *IEEE Trans. on Parallel and Distributed Systems*, 10(6):627–641, June 1999.
- [2] J. Allen and A. Christie. State of the Practice of Intrusion Detection Technologies. Technical Report CMU/SEI-99-TR-028, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, Jan. 2000.
- [3] S. Bagchi. *Hierarchical Error Detection in a Software Implemented Fault Tolerance (SIFT) Environment*. Ph.D. Dissertation, University of Illinois at Urbana-Champaign, Urbana, IL, Jan. 2001.
- [4] M. Dacier, Y. Deswarte, and M. Kaaniche. Models and Tools for Quantitative Assessment of Operational Security. In *Proc. of 12th IFIP Information Systems Security Conf. (IFIP/SEC'96)*, pages 177–186, May 1996.
- [5] Intel Corporation. *Intel Architecture Software Developer's Manual, volume 2, Instruction Set Reference*, 1999.
- [6] R. K. Iyer and D. Rossetti. Effect of System Workload on Operating System Reliability: A Study on IBM 3081. *IEEE Trans. on Software Engineering*, 11(12):1438–1448, Dec. 1985.
- [7] M. Kalyanakrishnam, Z. Kalbarczyk, and R. K. Iyer. Failure Data Analysis of a LAN of Windows NT Based Computers. In *Proc. of 18th IEEE Symp. on Reliable Distributed Systems*, pages 178–187, 1999.
- [8] B. Kantor. BSD Rlogin. *RFC 959*, Dec. 1991.
- [9] A. Mahmood and E. McCluskey. Concurrent Error Detection Using Watchdog Processors - A Survey. *IEEE Trans. on Computers*, 37(2):160–174, Feb. 1988.
- [10] R. Macion and K. Tan. Benchmarking anomaly-based detection systems. In *Proc. Int'l Conf. on Dependable Systems and Networks (DSN 2000)*, pages 623–630, June 2000.
- [11] G. Miremadi, J. Karlsson, U. Gunnefl, and J. Torin. Two Software Techniques for On-Line Error Detection. In *Proc. Int'l Symp. on Fault-Tolerant Computing (FTCS-22)*, pages 328–335, July 1992.
- [12] B. C. Neuman and T. Ts'o. Kerberos: An Authentication Service for Computer Networks. *IEEE Communications*, 32(9):33–38, Sept. 1994.
- [13] R. Ortalo, Y. Deswarte, and M. Kaaniche. Experimenting with Quantitative Evaluation Tools for Monitoring Operational Security. *IEEE Trans. on Software Engineering*, 25(5):633–650, Oct. 1999.
- [14] J. Postel and J. Reynolds. Telnet Protocol Specification. *RFC 854*, May 1983.
- [15] J. Postel and J. Reynolds. File Transfer Protocol (FTP). *RFC 959*, Oct. 1985.
- [16] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public key cryptosystem. *Communications of the ACM*, 21(2):120–126, Feb. 1978.
- [17] D. T. Stott, B. Floering, Z. Kalbarczyk, and R. K. Iyer. Dependability assessment in distributed systems with lightweight fault injectors in NFTAPE. In *Proc. IEEE Int'l Computer Performance and Dependability Symp. (IPDS'2K)*, pages 91–100, Mar. 2000.
- [18] J. Xu, Z. Kalbarczyk, and R. K. Iyer. Networked Windows NT System Field Failure Data Analysis. In *Proc. of IEEE Pacific Rim Int'l Symp. on Dependable Computing*, pages 178–185, Hong Kong, China, Dec. 1999.
- [19] T. Ylonen. The SSH (Secure Shell) Remote Login Protocol. *Internet-Draft (draft-ylonen-ssh-protocol-00.txt)*, Nov. 1995.