An Experimental Study of File Permission Vulnerabilities Caused by Single-Bit Errors in the SELinux Kernel Policy File

Michael Ihde, Tom Brown
Department of Electrical and
Computer Engineering
University of Illinois at Urbana-Champaign
Urbana, IL 61801
Email: {ihde tdbrown}@uiuc.edu

***Abstract*— System security is often analyzed and studied assuming error-free operation of the computer. Recently, due to increasing transistor densities and processing speeds, there has been renewed interest in the research of single-bit transient errors and the effect they have on availability and performance. We believe that it is also important to analyze security enforcement, specifically kernel level enforcement, in the face of single-bit transient errors.**

Error-injection experiments are used to quantify the impact that single-bit errors have on the Linux kernel with SELinux enhancements. The SELinux enhancements provide Linux with mandatory access controls, fine-grained access control lists, and role-based-access controls. These controls are meant to isolate processes and limit the power of users, including the root user. Any failure to enforce the security policy could allow a major security breach.

Error-injections are performed on the policy database as it is being loaded into the kernel. Tests are then run against a target file system to determine if the error created any security vulnerabilities. The results show that approximately 33% of the errors cause the policy to fail during loading. Of the remaining errors we found 8 which allowed one of our prohibited tests to succeed.

I. Introduction

Security research is largely focused on providing methods, procedures, and best-practices that will make our computer systems secure during error-free operation. There is no doubt that this approach is valuable and will continue to be valuable in the future. Still, we must not overlook the fact that errors can and do occur in computers and these errors do have impact this system. Recent work has indicated that transient single bit errors can significantly affect system downtime. Simultaneously, researchers in high-performance computing have begun to take great interest in single-bit errors for they are now a predominant factor affecting performance limits. This study attempts to explore security in the face of single-bit errors. Using the SELinux kernel as a target we have gained an initial understanding of how single-bit data errors can create vulnerabilities in the security mechanisms of an operating system.

The Linux kernel with SELinux enhancements developed by the National Security Agency was chosen to be the target of our the error injection study. SELinux was selected for many reasons: it has been widely adopted by most of the mainstream Linux distributors, it is fairly flexible, it has sufficient complexity to be interesting, it offers fine-grained mandatory access controls (MAC), and the source code is freely available. Other operating systems that offer some form of MAC would also be interesting targets for error-injection.

Errors were injected using a modified kernel. The modifications were not intrusive and only affected kernel operation when loading the policy at startup. Therefore, these modifications should have little effect on the outcome of the study. This method was selected for convenience and could be replaced in future studies by a non-intrusive error injector. As the policy is loaded into the system a specific bit is flipped. The location of this bit is a kernel parameter. This form of error most closely resembles a fault that occurred on the disk image of the policy database but may also represent a fault in the memory or processor under certain conditions.

Section II reviews related work, section III presents SELinux, section IV presents examples of vulnerable code, section V provides a detailed description of our experimental approach, section VI presents our results, and finally the conclusion is presented in VII.

II. Related Work

Single-bit faults have been shown to have a drastic effect on system reliability, especially in high-altitude or extraterrestrial environments where the error rates are the highest due to cosmic rays. This is becoming more of a problem in standard operating environments, as when semiconductor densities increase and the supply voltages decrease the soft error rates increase. Results from [1] indicate that one 4Mbit DRAM will suffer from approximately 6000 Failures in Time (FIT), or failures in a billion hours, if no error correction is performed in the memory.

Using Error Correcting Codes (ECC) only serves to reduce the problem. A portion of errors will still go undetected in the presence of ECC. Faults in the processor or the communication are not correctable by ECC in the memory. In [2] a system

with 1GB of memory composed of 64Mbit DRAM cells had 3435 FIT when using standard ECC. As stated in [3] this is equivalent to around 900 errors in 10000 machines over 3 years. If only 1% of these errors created a security vulnerability it would be an issue worth fixing. For example, Google (TM) is known to use at least 15,000 commodity-class PCs to provide their web search service [4]. Given a 1% vulnerability rate there would be 13 vulnerabilities in a 3 year period. More devastating attacks could occur if an attacker were to successfully attack only one of these machines.

The Linux kernel is often used in reliability studies to characterize the effect of errors on the system. Some have injected errors using a customized user mode kernel [5] [6] while others have injected errors directly into a running kernel [7]. In [7] up to 47% of the injected errors did not manifest themselves in a detectable fashion. It is conceivable that a portion of these errors may have caused security vulnerabilities.

In [8] single-bit faults were injected to induce control flow errors in sshd and ftpd, two commonly used Internet services. This was the first paper to explicitly show the potential for single-bit faults to cause a security vulnerability in real networked applications. The authors chose to perform selective exhaustive injection to reduce the number of injected errors ; reducing the time and computational resource requirements. Approximately 2-8% of the code was injected with errors. The selections were made by identifying portions of the code that related to user authentication. Errors were then injected onto every branch instruction, creating control flow errors. A majority of the errors caused crash failures (43-63%), but a portion (1-2%) created permanent vulnerabilities.

Single-bit errors do not only represent a theoretical security threat. Virtual machines like the Java Virtual Machine and the .NET architecture are susceptible to real vulnerabilities caused by single-bit errors. An attacker using a heat source, such as a lamp, can increase the error rate enough in a smartcard [9] to perform the attack in a reasonable amount of time. Earlier research required sophisticated techniques, a high degree of skill, sophisticated equipment and a decent amount of time [10] [11].

Our study follows on the research performed by [8] and extends it to kernel level security implementations. All previous studies have only performed security investigations in user space programs, with virtual machines occupying the gray area between kernel and user space. Fault-tolerant security is an emerging concern as commodity PCs are replacing their more reliable predecessors, mainframes. In these environments, hardware fault tolerance is eschewed in favor of simple redundancy, increasing the likelihood of soft errors. Companies such as Google already employ thousands of cheap, unreliable PCs in a massive redundant configuration [4]. This method can provide traditional fault tolerance, but may not provide adequate fault-tolerant security. To the best of our knowledge this is the first study to perform a fault-tolerant security study by using error injection on the SELinux kernel.

III. SELinux

SELinux is a series of enhancements to the standard Linux kernel developed by the National Security Agency. Originally SELinux was distributed as a patch to a regular kernel. The latest stable version of Linux (2.6) now contains SELinux as a regular option. Many Linux distributors are currently shipping or plan to ship products with SELinux enabled out of the box.

The SELinux enhancements provide Mandatory Access Controls (MAC) to Linux through the Linux Security Module (LSM) framework. The regular file permissions/user login in Linux represent Discretionary Access Controls (DAC). In a DAC model a compromised program running as the root user would have complete access to the system. Under the MAC model processes are granted/denied permissions using more information than just the user identity. The SELinux model defines permissions using subjects (users, programs, processes) and objects (files, devices).

SELinux allows flexible and fine-grained control over any object access at the cost of additional complexity. Under SELinux a processes such as sendmail (which typically is run as the root user) could be limited access to only the resources necessary to perform it's task. If it were compromised the attacker would only gain access to these resources.

IV. Examples of Vulnerable Code

Using the source code in Figure 1 we will illustrate the effect a single-bit data error can have on SELinux policy enforcement. There are four variables of interest in Fig. 1: denied, requested, allowed, and SELinux_enforcing. All the variables are unsigned 32-bit integers. The allowed variable comes from the policy database through the access vector cache (AVC). A single-bit error in either the policy database or the AVC could hit the allowed variable. The enforcing variable is modified via the SELinux virtual filesystem and allows the system administrator to enable or disable SELinux permission enforcement. Both of these variables are susceptible to single-bit errors and will be described in greater detail in the following two subsections.

The encoding used for requested makes it safe from single-bit errors. In nearly all cases the requested variable contains all zero bits except for a single one bit. A single-bit error will either deny an authorized user (requested has to be non-zero and match allowed) or have no effect at all. If a single bit error were to corrupt the denied field it could create a security vulnerability. However this would be a rare occurrence because the denied value is updated and used within a few instructions. Any transient errors latent in the memory location would be removed when the data was written to memory.

A. Single Bit Faults Affecting the Enforcing Variable

A single-bit error in the first bit of the SELinux_enforcing variable will allow all SELinux permission checks to pass, in essence falling back to the regular DAC of the Linux. If this occurs, it will remove all SELinux security checks, creating a massive vulnerability if the system was dependent of SELinux security. Furthermore, the system administrator will be given

```
denied = requested & ~(ae->avd.allowed);

if (!requested || denied) {
  if (selinux_enforcing) {
    rc = -EACCES; goto out;
  } else {
    ae->avd.allowed |= requested;
    goto out;
  }
}
```

Fig. 1. Source Code

```
avc:  denied  { append } for  pid=850
exe=/bin/bash name=passwd
dev=ubda ino=28951
scontext=root:staff_r:staff_t
tcontext=system_u:object_r:etc_t
tclass=file

requested = 0x00000200
ae->avd.allowed = 0x00022053
```

Fig. 2. Append to /etc/passwd by staff_r

a false sense of security because the AVC denial messages will still be generated. Luckily, only one bit is vulnerable and so it is very unlikely that an error will affect this bit before the system crashes, even when the bit-error-rate is abnormally high.

B. Single Bit Faults Affecting the Allowed Variable

A single-bit error in the allowed variable will have the greatest impact on the security of the system. If the error causes a 0 to 1 transition then additional permissions are granted, if the error causes a 1 to 0 transition then a given permission will be denied, the later being merely an inconvenience. The allowed variable may be corrupted while it is in the policy database or the AVC. It represents the bitwise OR of all the allowed permission for a particular security domain. For example, the append permission is 0x00000200. This encoding scheme ensures that a single-bit error can cause a security vulnerability under the right conditions. We can see a typical permission denial in Fig. 2. In this case the root user under the role staff_r attempts to append data to the protected file, /etc/passwd. If an error affected the 9th bit in the allowed field then permission will be granted to the user. It is difficult to determine the window of vulnerability because it is dependent on the location of the error. If the error were in the policy database it would persist until a policy reload. If the error were in the AVC it would persist until the cache element was refreshed or overwritten.

V. Experimental Approach

Given the complexity of the SELinux code it would have been difficult to determine all locations where a single-bit error could cause a vulnerability by inspection of the source. We used an experimental approach to exhaustively inject errors into the policy database as it was being loaded.

A. User-Mode Linux

The centerpiece of our experimental approach is User-Mode Linux. User-Mode Linux is a set of enhancements that can be compiled into a standard Linux kernel. These enhancements allow the Linux kernel to run as a user-mode program. Nearly all of the original kernel code is used, with only a small portion being modified to support userspace operation. Therefore a user-mode kernel should have nearly identical behavior to that of a regular kernel. In many ways usermode Linux is similar to an x86 emulator, such as VMware. However, usermode Linux does not emulate an individual processor, but simply utilizes the host kernel to act on the hardware. This allows usermode Linux execute with considerably less overhead. It also would have been possible to perform our experiments directly on the host kernel, but User-Mode Linux provides many benefits over this traditional method. If the kernel hangs it can be detected without requiring a hardware watchdog timer, file system corruptions can easily be recovered, tests can be parallelized across many computers, and the kernel itself can be debugged using standard debugging tools such as gdb.

B. Client and Server Operation

We created a simple infrastructure to boot the user-mode kernel with a different error each time. A server keeps a list of every error to try and it collects the results of the boot with each error. The client, which may run on many hosts in parallel, requests a job from the server, sets up the file systems for the user-mode kernel, and boots the kernel. Once the kernel has started, the SELinux policy is loaded with the requested error. A script running inside the kernel then attempts to violate the policy by making several prohibited requests. The client gives the script 30 seconds to finish and shutdown the user-mode kernel, after which time the system is considered hung and killed. The client then checks the file system to see if SELinux enforced its policy in spite of the error. Finally the client gathers all the results, returns them to the server, and requests a new job.

C. Error Model

Error injections are performed on the policy database as it is being loaded into the kernel. These errors most closely represent an error on the policy image that is stored on disk. An error on the disk is a permanent error that will exist until the policy image is recreated by the administrator. If the policy successfully loads than the error can represent a wider range of faults including: faults in the disk image, faults in the main memory, faults in the cache, and faults in the processor.

D. Error Injection

Choosing the method used for error injection required certain trade-offs. The most recent policy distributed for SELinux

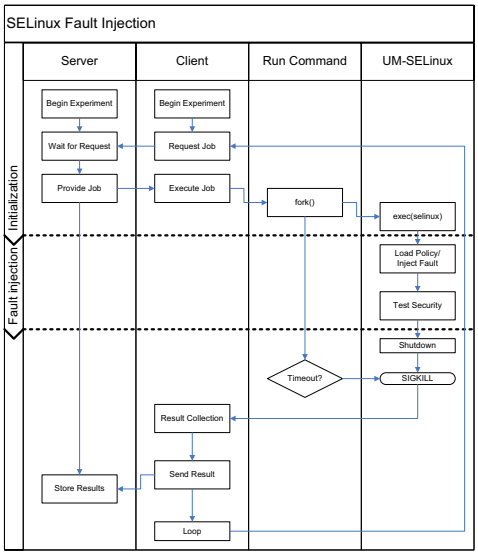

Fig. 3. Fault Injection Framework

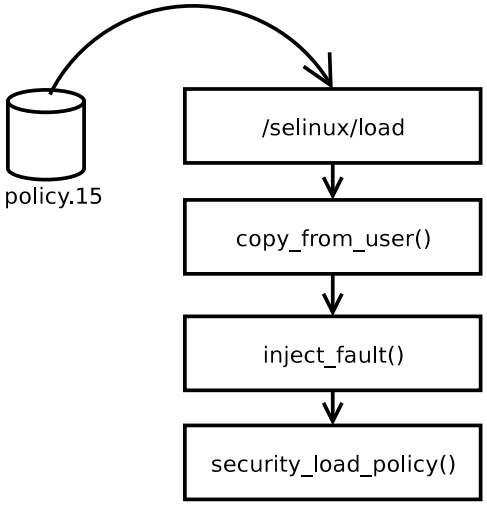

Fig. 4. Fault Injection Location

```
static inline void inject_fault(
  char* data, int count)
{
  unsigned char fault_mask =
      0x01 << fault_bit;

  *(data + fault_byte) =
      *(data + fault_byte) ^ fault_mask;
}
```

Fig. 5. Fault injector routine

creates a 500kB policy image. Performing exhaustive testing on a policy this large would require significant computing resources. Given that it would be difficult to determine a priori the sections of the policy database that were most relevant to security, we could not resort to selective exhaustive injection. If, instead, we chose random injection we would risk the chance that the injected errors would create vulnerabilities in the policy that did were not explicitly tested or have no effect at all while skipping over potentially interesting bits. Our experimental method is a trade-off between the two extremes. To make exhaustive injection feasible the policy was simplified by removing all unnecessary rules. This reduced the policy to approximately 18kB.

The error injector was implemented in the kernel code by modifying the selinuxfs.c file. Figure 5 shows an abbreviated listing of the fault injector code. The variables fault_bit and fault_byte are passed on the command line as kernel parameters. The location of the error can then be changed by running the kernel with different parameters. After the policy has been read from the file system it is copied to kernel memory space and then the error is injected into the desired byte. After the error is injected the policy is passed to the normal SELinux routines.

The policy is loaded into kernel space via the SELinux virtual file system. This virtual file system is typically mounted at /selinux and provides a device node to load the policy from user space. Under normal conditions the init process is responsible for loading the policy into the kernel. To speed execution of the experiments we have removed the normal init process and replaced it with a special init program that loads the policy, perform a set of security tests, and then attempts to shutdown the kernel. Replacing the standard init reduced execution time on a Pentium II 400MHz computer to under 10 seconds compared to over a minute for a regular boot. Furthermore, the only process running in our test environment is the init program allowing us to conduct controlled security tests with faults on each bit in the policy in turn.

E. Test Policy

As mentioned above, our error injections were performed on a simplified policy. The base SELinux policy contains 3 users, 5 roles, 313 types, 30 classes and 19741 rules. The simplified policy used for our experiment has only 1 user, 2 roles, 162 types, 30 classes, and 301 rules. Reducing the rule set did more than just reduce the file size. It also provided a simpler environment to test, hopefully increasing the chance that our error injections would create vulnerabilities in the areas we were checking.

Our simplified policy follows a basic *deny all* strategy to the kernel_t domain, in which our test init runs. The only permissions explicitly granted are those required to start the test. All other permissions are denied. This is clearly very simplified and many vulnerabilities will be prevented because most operations will perform multiple unique security checks. Our policy is not representative of one that could be found in a real system which would be larger and more complex and thus more susceptible to errors in design.

VI. Results and Discussion

A total of 144818 errors were injected and tested. This resulted in approximately 32.5 million lines of output. We

TABLE I
RESULTS OF INJECTIONS

Errors That Caused Vulnerabilities	8
Errors That Had No Measured Security Impact	96770
Errors That Failed Policy Load	48040
Total Errors Injected	144818

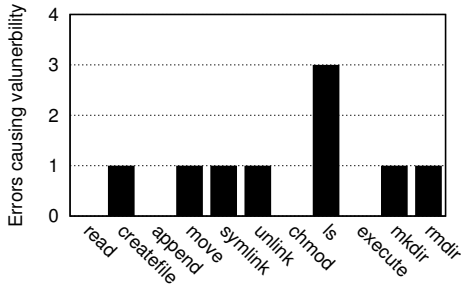

Fig. 6. Vulnerability Location

parsed the output to look for runs with abnormal behavior.

A significant portion of the errors (33%) caused the policy to fail during the loading. This may or may not represent a security vulnerability, depending on the behavior of the init program. For our experiments, the kernel was allowed to continue with no policy loaded, allowing full permissions. The regular init distributed with SELinux will halt the kernel with an error message preventing these errors from causing a vulnerability.

Our test script checked the enforcement of restrictions on 11 different types of actions, each of which should have been blocked by the SELinux policy. The 11 actions focused on file and directory restrictions: file read, file create, file append, file move, soft link creation, file unlink, file chmod, file execute, directory list, directory create, and directory remove. Unlike previous studies, which tested authentication or privilege escalation, there is not a single operation that can be performed to evaluate SELinux security.

The results are presented in Table I and Figure 6. The relatively small number of vulnerabilities which produce errors seems to indicate that the problem in not very serious. Perhaps this is true, but given the simplifications we required and a few experimental flaws there may be a more significant number of interesting errors.

A. Difficulties and Problems

One of the flaws which considerably affected our study is we started with a policy that allowed file read and execute operations during an error-free run. We did not realize this mistake until it was too late to correct. An interesting result from those portions of the test is that 1.3% of errors successfully injected into the policy *prevented* the file read access, which should have been granted according to the policy. Clearly this is not a vulnerability, but it could create a denial of service problem.

We believe that the results we have presented are biased against allowing a vulnerability because many of the operations that perform modifications to the target filesystem require multiple permissions to be granted to be successful. In the cases where only one of the permissions was allowed we would see no end effect, and thus would not notice any weakness. This is good for system security in general, but makes objective measurement difficult.

Part of the SELinux policy specifies what actions it should output for auditing. When conducting this experiment we only enabled auditing of permission checks when access is denied. In future studies we shall also enable auditing of checks where access is allowed, giving us a more complete picture of why a specific error denies or allows some action.

While we shrank the policy in an attempt to simulate a fault into every possible bit we ran out of time before we could complete every run. We ran with each error once but approximately 3.8% of the runs were lost due to possible User-Mode Linux crashes. We knew our version of User-Mode Linux would occasionally crash independently of the presence of an error, destroying the output of a run, but did not have time to try every error that experienced a crash. We also did not expect a bug in the kernel of our host machine which required a reboot every three hours. We hoped to have access to more than two machines and future experiments should complete more quickly once we get access to them. Complete results from all runs will be available within a few days.

VII. CONCLUSIONS AND FUTURE DIRECTION

Not surprisingly, we have shown that a single bit flip can create a serious vulnerability in a computer system. A single bit flip in the SELinux policy can cause more than one action to be allowed which an error free policy would deny. Of the 144818 faults we simulated we found 8 that corrupted the policy enough to allow at least one of our prohibited accesses to succeed.

A complete policy which allows all the access needed for a system to operate correctly while denying most other access will be complex. There are a very large number of ways one could attempt to violate such a policy, making complete testing very difficult. Building a policy that assures some reliability when faced with single bit errors is even more difficult. We have shown that a single bit error in SELinux does not necessarily change a single subject's access to a single object. The rate at which these errors do have any effect on the system security enforcement is very low. A bit flip in memory is much more likely to *prevent* access, rather than deny it. This was observed because many of our tests needed multiple access (getattrib for directory and file in addition to actual test). Extremely security-sensitive administrators, such as those who use SELinux, should understand the risk.

Continuing this work we will restart injecting errors into our shorter policy file, correcting the mistakes described in Difficulties and Problems. Because the number of errors which cause a measurable vulnerability is small we will trace back to the cause of each of them in the source code. We would

like to compare these results to those generated by a conventional policy injected with random errors. We also intend to inject errors directly into the text and data sections of a totally unmodified user-mode kernel using the ptrace interface. We are also interested in having someone familiar with the implementation of SELinux attempt to analyze it for single bit error weaknesses based on their evaluation of the source code, providing a comparison to our experimental method.

REFERENCES

[1] J. Ziegler *et al.* (2003, May) Ibm experiments in soft fails in computer electronics (1978-1994). [Online]. Available: http://www.research.ibm.com/journal/rd/401/curtis.html

[2] T. J. Dell. (1997, Nov.) A white paper on the benefits of chipkill-correct ecc for pc server main memory. [Online]. Available: http://www-1.ibm.com/servers/eserver/pseries/campaigns/chipkill.pdf

[3] A. Messer *et al.* (2001, Mar.) Susceptiblity of modern systems and software to soft errors. [Online]. Available: http://www.hpl.hp.com/techreports/2001/HPL-2001-43.pdf

[4] L. A. Barroso, J. Dean, and U. Holzle, "Web search for a planet: The google cluster architecture," *IEEE Micro*, pp. 22–28, Mar./Apr. 2003.

[5] K. Buchacker and V. Sieh, "Framework for testing the fault-tolerance of systems including os and network aspects," in *Proc. IEEE Sixth International High-Assurance Systems Engineering Symposium (HASE'01)*, 2001, pp. 95–105.

[6] V. Sieh and K. Buchacker, "Umlinux - a versatile swifi tool," in *Proc. of the Fourth European Dependable Computing Conference*, Toulouse, France, Oct. 2002, pp. 421–432.

[7] W. Gu, Z. Kalbarczyk, R. Iyer, and Z. Yang, "Characterization of linux kernel behavior under errors," in *Proc. IEEE International Conference on Dependable Systems and Networks (DSN'01)*, San Francisco, CA, June 2003, pp. 459–468.

[8] J. Xu, S. Chen, Z. Kalbarczyk, and R. Iyer, "An experimental study of security vulnerabilities caused by errors," in *Proc. IEEE International Conference on Dependable Systems and Networks (DSN'01)*, Goteborg, Sweden, July 2001, pp. 421–432.

[9] S. Govindavajhala and A. W. Appel. (2003, May) Using memory errors to attack a virtual machine. [Online]. Available: http://www.eecs.harvard.edu/cs253/papers/Govindavajhala03MemoryErrors.pdf

[10] R. Anderson and M. Kuhn, "Tamper resistance - a cautionary note," in *Proc. of The Second USENIX Workshop on Electronic Commerce*, Oakland, CA, Nov. 1996, pp. 1–11.

[11] O. Kommerling and M. Kuhn, "Design principles for tamper-resistan smartcard processors," in *Proc. of USENIX Workshop on Smartcard Technology*, Chicago, IL, May 1999, pp. 9–20.