Taxonomy of Linux Kernel Vulnerability Solutions

Serguei A. Mokhov
Computer Security Laboratory
Concordia Institute for
Information Systems Engineering
Concordia University,
Montreal, Quebec, Canada
Email: mokhov@ciise.concordia.ca

Marc-André Laverdière
Computer Security Laboratory
Concordia Institute for
Information Systems Engineering
Concordia University,
Montreal, Quebec, Canada
Email: ma_laver@ciise.concordia.ca

Djamel Benredjem
Computer Security Laboratory
Concordia Institute for
Information Systems Engineering
Concordia University,
Montreal, Quebec, Canada
Email: d_benred@ciise.concordia.ca

***Abstract*—This paper presents the results of a case study on software vulnerability solutions in the Linux kernel. Our major contribution is the introduction of a classification of methods used to solve vulnerabilities. Our research shows that precondition validation, error handling, and redesign are the most used methods in solving vulnerabilities in the Linux kernel. This contribution is accompanied with statistics on the occurrence of the different types of vulnerabilities and their solutions that we observed during our case study, combined with example source code patches. We also combine our findings with existing programming guidelines to create the first security-oriented coding guidelines for the Linux kernel.**

***Index Terms*—Linux kernel, Software Vulnerabilities, Vulnerability Remedial, Vulnerability Solutions Taxonomy**

I. Motivation

Linux solutions for both servers and desktops are increasing in popularity and notably so in the developing world as well as virtualization. Today, systems running on the Linux operating system kernel represent 12.7% of server market and estimated 3%-6% of desktop systems [32], [11]. As Linux is becoming an important actor in the computing world, its security becomes paramount. Similarly to other Unix and Unix-like systems, there is a wide variety of software (desktop, shells, services, etc.) that can be chosen by the administrator from various vendors and the security of a Linux-based system thus relies on the security of a wide set of applications, libraries, programming languages, etc. However, what is unique to a Linux system is the Linux kernel itself. Since the kernel handles file and network operations, access control, resource allocation, etc., its correct functioning is an important part of the overall system security.

We performed this case study on the Linux kernel due to its open-source nature, its popularity, and its development model. This popularity has created a greater interest in scrutinizing the source code for potential vulnerabilities, yielding substantial amount of vulnerability reports to work from. Our study examined the solutions to the vulnerabilities reported in 290 CVE (Common Vulnerabilities and Exposures referred to from [21]) advisories, in the 2.4.x and 2.6.x version families. From these solutions, we drew a classification scheme and computed statistics related to the usage of these families of solutions.

Our major contribution is the classification of the security solutions used for the improvement of the Linux kernel, and C programs in general, as well as statistics on their relative importance. We also introduce a new methodology to track the patch solving a security issue based only on the contents of the security advisory.

The paper is organized as follows: we examine previous work that was done regarding Linux and C security in Section II, followed by a description of the methodology used in order to obtain the solutions to the vulnerabilities in Section III. Afterwards, in Section IV, we show our results, notably classification of the vulnerability solutions we define, as well as statistics related to the occurrence of certain vulnerabilities and their solutions in our data set. In Section V, we illustrate the vulnerability solutions with patches from our data set. In Section VI, we introduce our guidelines for Linux kernel development based on our findings. Finally, in Section VII, we conclude.

II. Related Work

The security of Linux has been covered at many levels in the IT industry. The level that drew the more ink so far has been related to administrative tasks. For example, [12], [19] describe the security features of Linux as well as how to properly configure Linux-based systems. Other works approach the security of Linux systems from the viewpoint of the programmer. For example, Wheeler [33] explains proper coding practices in both generic and language-specific cases. Furthermore, some studies on the security of the Linux kernel done in the past incorporated useful statistics on the preponderance of certain types of software vulnerabilities [5], [25], [34].

The world of software vulnerabilities in C and C++ has been extensively studied [26], [14], [33], etc. This literature will often list various forms of software vulnerabilities, explain, and illustrate them. They will also often propose a method that can be used to remediate the vulnerability. When turning to coding guidelines or code review guidelines [27], [29], [20], [30], [1], [17], we see a need for simple and relevant documentation. In some cases, we can find literature [12], [16] that will describe how we can attack Linux systems as well as countermeasures to such attacks.

The open-source community created some high-security variants of Linux for environments requiring it. A first example is the Linux Security Modules architecture [35], especially

known for SELinux [28], which implements strong security policies, and LIDS [15], which offers intrusion detection capabilities. Another option available is Owl [23], a distribution that ensures that the software included runs with minimal privileges and fail-safe defaults which also offers a modified version of the kernel.

However, as far as we know, there is no study that was done on the techniques practically used by developers to remedy Linux kernel vulnerabilities and their relative importance.

III. Approach

In this section we summarize the methods we used to find the remedial methods for given vulnerabilities. Over the course of roughly two years as a part of the research course work, master's research work [24], and general interest we gathered the statistics of common vulnerability solutions, taxonomy of which we present in this paper, covering most Linux kernel vulnerabilities from 2002 to 2007.

We used a spreadsheet program[1] to keep track of vulnerability numbers, version(s) of the Linux kernel, the classification of the vulnerability, links to the solutions, notes on the methodology used to obtain the solution, solution summaries, and the vulnerability solutions' classification in the comma-separated values (CSV) format. This spreadsheet was then processed using a Perl script which generated the statistical information for the counts, tables, and the relevant graphs.

We will now look in more depth at the methodology used to obtain our results. We first chose a reliable data source for Linux software vulnerabilities and extracted relevant vulnerability reports, then used the vulnerability reports as a starting point to find the patch associated. In parallel and after this last activity, we established a classification of the solution types we observed and assigned one or more classification to each of the patches found. We finally developed an automated tool that extracted statistics from the data set we were using.

For the extraction of relevant vulnerabilities, we decided to use the National Vulnerability Database (NVD) [4], [21] for a list of vulnerabilities for its official nature, its comprehensive contents and integrated statistics capabilities [22]. We extracted all the unique vulnerability numbers (the older style of CAN – YYYY – NNNN and the current style of CVE – YYYY – NNNN) for all 2.4 and 2.6 Linux kernel versions using the NVD web interface. This gave us a relatively shortlist of 290 advisories to investigate between years 2002 and 2007.

The second step of our research was to track the precise patch that solved each vulnerability. This task was non-trivial, as no comprehensive database kept track of this information and that the Linux kernel bug-tracking system contained no version 2.4 information and would rarely return any results on CAN/CVE numbers. Thus, we needed to resort to other methods to find patches we were seeking.

We tried to use, as much as possible, the information that was provided directly in the advisory. Often, it included a link to the revision control systems (BitKeeper [2] and GIT [36]), giving us directly the patch related to the problem. Sometimes, the advisory would lead us to Linux vendors' advisories and bug tracking systems that lead to a patch addressing the vulnerability. The advisories would also refer to Bugtraq [3] emails that announced the vulnerability, which sometimes included a patch to the vulnerability. At other times, the best information that we could find from the advisory was the function name, source code file or driver affected, as well as the vulnerable versions. This allowed us to search the patch of the first unaffected version for this precise information and find the solution to the security issue.

We also found that the vulnerability number, expressed under the CAN/CVE schemes, could often be found in vendors' bug tracking systems, maintainer mailing lists, maintainers' management systems [8], enthusiasts' websites [13] and newsgroups [6]. This information often leads us, directly or indirectly, to the patch for the given vulnerability.

The Linux kernel project, for each version, includes a `ChangeLog`. The changelog sometimes would include the CVE/CAN number, or have a comment that allowed us to conclude that it was related to the vulnerability investigated. The commit comment or GIT commit number was then queried for in the BitKeeper [2] archives.

As we were finding the vulnerability solutions, we also examined them and noted a summary of the code changes we observed. Broad categories quickly emerged as we were doing this activity, which we started allocating to solutions that were matching said categories. In order to ensure reasonble consistency and proper classification, the authors refined the classification and each category's definition, and then collectively re-examined most of the patches to ensure that they were correctly classified. Afterwards, we used the earlier mentioned Perl script in order to regroup and count the occurrences of each vulnerability and solution and generate all the statistical figures and table present in the paper.

IV. Results

In this section, we first summarize some information about the nature of the vulnerabilities examined, then proceed to describe our classification, to finally offer some statistics we drew from our vulnerability solutions data.

A. Vulnerability Classification

We used the classification directly from CERT [4], which was included in every vulnerability report. Most of the categories were defined in [5], which we took the freedom of citing and expanding as follows:

Input Validation Error – "failure to recognize syntactically incorrect input" from the interactive user or from a module, process, or function. This category includes subcategories such as buffer overflows and boundary condition errors; however, due to a large number of vulnerabilities in these two subcategories, we mention them separately.

Buffer Overflow – failure to ensure that a computation does not write information outside the intended address space.

[1]Eventually we hope to migrate to a more flexible and accessible form to manage this database and possible to make it available on-line.

Boundary Condition Error – failure to ensure that the acceptable domain or range of data is respected.

Access Validation Error – allowance of "an operation on an object outside its access domain."

Exceptional Condition Handling Error – "system failure to handle an exceptional condition generated by a functional module, device, or user input."

Environmental Error – "an interaction in a specific environment between functionally correct modules."

Configuration Error – "a system utility installed with incorrect setup parameters."

Race Condition – "an error during a timing window between two operations."

Design Error – improper design of the security mechanism or implementation not satisfying the design.

Nonstandard – errors not matching any other description or "unknown" errors per CERT.

B. Remedial Classification

While gathering and observing our data, we established 13 categories of solutions that were used by Linux kernel maintainers, which classify, define, and illustrate as follows:

Change of Data Types refers to the use of more appropriate data types (such as unsigned instead of signed) or non-static when necessary.

Precondition Validation refers to ensuring that an operation's set of preconditions are met before proceeding with its execution. This is typically done by checking for a valid variable values and range and returning an error code if the conditions are not met. Additionally, this may include the improvement of an existing precondition check, for example by correcting an existing boundary, or adding a missing upper/lower one.

Ensuring Atomicity refers to code modifications that guarantee non-concurrent execution of critical sections, typically by adding locks, semaphores, etc.

Error Handling adds or improves the verification and handling of error conditions. This is typically done by checking the return value of a function call or adding statements in previously existing error handling code. Please note that error handling is most easily distinguished from precondition validation by the fact that it typically occurs within the body of the function, whereas precondition validation occurs at the beginning of a function.

Zeroing Memory is about ensuring that selected memory contents are overwritten by filling zero values.

Freeing Resources guarantees that resources such as memory and timers are being freed after usage.

Input Validation is the practice of validating or modifying input data in order to ensure it complies with a safety policy.

Capability Validation allows the execution of an operation only after a successful access control check.

Fail-Safe Default Initialization is the setting of a guaranteed safe value for variables before an operation requiring its use. This includes the assignment of a default value at declaration time and ensuring null-termination of strings.

TABLE I
VULNERABILITY TYPE DISTRIBUTION

Vulnerability Types	**Count**	**%**
Design Error	91	28.35
Input Validation Error	55	17.13
Exceptional Condition Handling Error	37	11.53
Buffer Overflow	30	9.35
Boundary Condition Error	29	9.03
Race Condition	25	7.79
Access Validation Error	18	5.61
Nonstandard	15	4.67
Not classified	11	3.43
Environmental Error	6	1.87
Configuration Error	4	1.25
Total	321	100.00

Protection Domain Enforcement guarantees that the data operated on, or the execution flow, is in the appropriate security domain. For example, the programmer can change a kernel reference to a buffer in the user space, which is unsafe, to a copy into the kernel space.

Redesign is often significant code changes, from refactoring to the change of a provider package, including the introduction of new or modification of the existing APIs.

Other solution that could not fit elsewhere in our classification and that was not observed enough (twice or less) to deduce a category. It includes adding function pointers in function tables, changing a buffer size, safe casting, safer pointer arithmetic, error reporting, loop termination, low-level assembly corrections, etc.

Unresolved are flaws to which a solution was not available yet at the time when the statistics was gathered.

C. Statistical Analysis

Of the 321 vulnerabilities to date[2] we found 339 solutions. We saw that design error, input validation error, and exceptional condition handling error were dominating the errors at the root of the vulnerability advisories (Table I). A corresponding diagram is in Figure 1.

In our analysis, we observed that precondition validation, error handling, and redesign were dominating the solutions (see Table II and the corresponding diagram in Figure 2). The numbers purposefully do not match the number of advisories evaluated (here 290 CVEs), as one advisory can contain multiple vulnerabilities, each potentially showing multiple methods used together to remedy it.

The additional statistics in Figure 3 lists counts of number of vulnerabilities in the kernel for 2.4, then for 2.6, and vulnerabilities that spanned across both branches.

V. EXAMPLE PATCHES

In this section, we illustrate each remedial method via a patch. The patch file format is a standard way to represent changes between two versions of the file [18].

[2]End of November 2007.

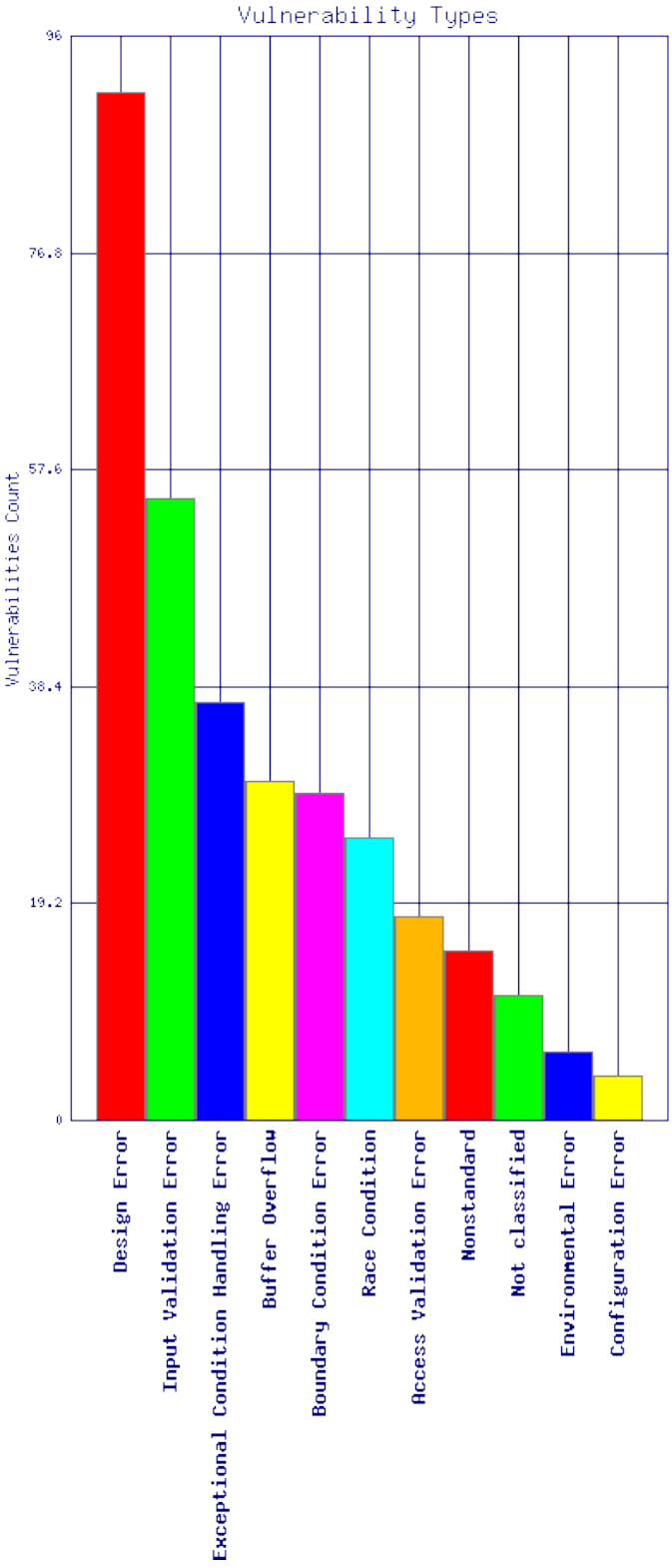

Fig. 1. Sorted Vulnerability Types Observed

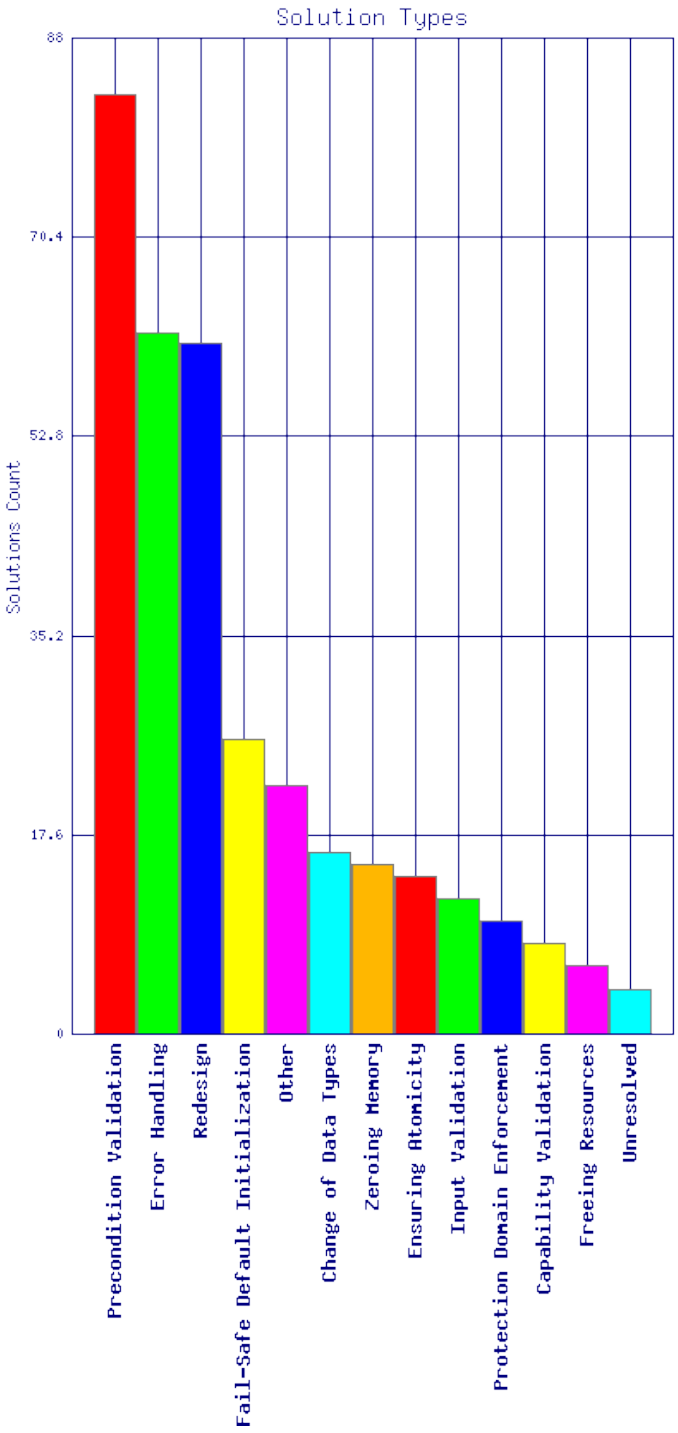

Fig. 2. Sorted Solution Types

A. *Change of Data Types*

In many cases, the data type used was not appropriate for the data representation needed or the concurrency scenario.

TABLE II
VULNERABILITY SOLUTION DISTRIBUTION

Remedial Types	Count	%
Precondition Validation	83	24.48
Error Handling	62	18.29
Redesign	61	17.99
Fail-Safe Default Initialization	26	7.67
Other	22	6.49
Change of Data Types	16	4.72
Zeroing Memory	15	4.42
Ensuring Atomicity	14	4.13
Input Validation	12	3.54
Protection Domain Enforcement	10	2.95
Capability Validation	8	2.36
Freeing Resources	6	1.77
Unresolved	4	1.18
Total	339	100.00

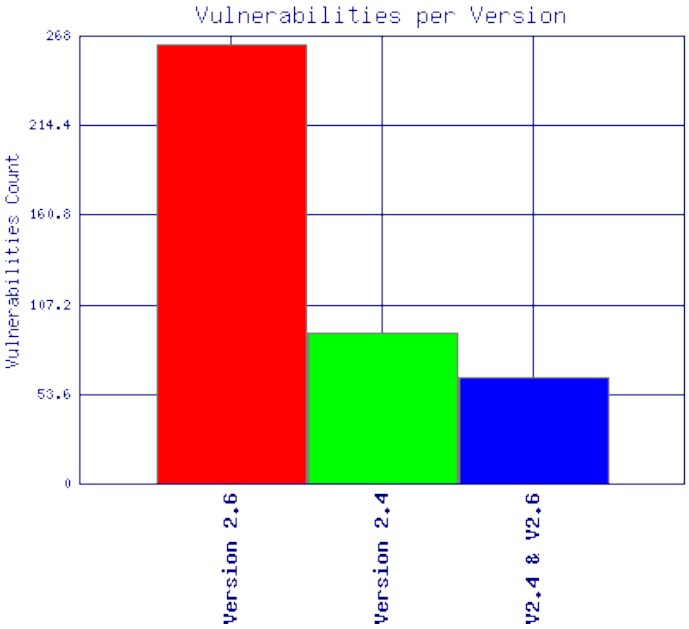

Fig. 3. Number of Vulnerabilities Found Per Kernel Branch

Typically, the change is from signed to unsigned types, or from static to non-static. The Listing 1 shows how a vulnerability was remedied by making a variable non-`static`, avoiding its sharing between threads.

B. Safe Casting

Some casting operations, such as integer promotions, are generated automatically by the compiler. In some cases, the casting can be inappropriate, especially when working on legacy code for new 64-bit platforms. The maintainer must sometimes add explicit casts and variables of the appropriate type in order to ensure that the data will be of the appropriate form. The listing 2 shows how a vulnerability was remedied by using a reference to the desired value of a safe data type, instead of relying on the compiler to correctly cast the pointer dereference.

```
{
  enum ip_nat_manip_type maniptype,
  const struct ip_conntrack *conntrack)
  {
-     static u_int16_t port, *portptr;
+     static u_int16_t port;
+     u_int16_t *portptr;
      unsigned int range_size, min, i;

      if (maniptype == IP_NAT_MANIP_SRC)
  }
}
```

Listing 1. Example of Change of Data Type by Removing Static Assignment for CVE – 2005 – 3275

```
    if (!state->pri_unat_loc)
      state->pri_unat_loc = &state->sw->ar_unat;
    /* register off. is a multiple of 8, so the
        least 3 bits (type) are 0 */
-   s[dst+1] = (*state->pri_unat_loc - s[dst]) |
    UNW_NAT_MEMSTK;
+   s[dst+1] = ((unsigned long) state->pri_unat_loc
    - s[dst]) | UNW_NAT_MEMSTK;
    break;

    case UNW_INSN_SETNAT_TYPE:
```

Listing 2. Example of Safe Casting Remedial to CVE – 2004 – 0447

C. Precondition Validation

Any function typically has a few implicit and/or explicit preconditions for its proper operation. These preconditions are not always validated, leaving room for potential vulnerabilities. A programmer must thus make those preconditions explicit and ensure that they are validated before the function's body is executed. The Listing 3 shows how a vulnerability was remedied by ensuring the validity of the `len` parameter.

```
     if (!len)
         return addr;

+    if ((addr + len) > TASK_SIZE || (addr + len) <
    addr)
+        return -EINVAL;
```

Listing 3. Example of Improved Precondition Validation for CVE – 2004 – 0003

D. Ensuring Atomicity

Some code must be inherently thread-safe, because of the nature of operating systems requiring concurrent execution of certain functions. As such, a maintainer must ensure proper locking of critical sections before executing them (or merely obtaining a reference to sensitive data structures), as well as ensuring the proper management of timers when some are involved. The Listing 4 shows how a vulnerability can be remedied by ensuring the locking of a data structure before manipulating it through a pointer.

E. Error Handling

In many cases, the error status of some functions is not checked, or the handling of the error is not appropriate in

```
  struct ebt_chainstack *cs;
  struct ebt_entries *chaininfo;
  char *base;
- struct ebt_table_info *private = table->private;
+ struct ebt_table_info *private;

  read_lock_bh(&table->lock);
+ private = table->private;
  cb_base = COUNTER_BASE(private->counters, private
      ->nentries, cpu_number_map(smp_processor_id())
      );
  if (private->chainstack)
```

Listing 4. Example of Atomicity Guarantee for CVE – 2005 – 3110

context. Programmers must thus ensure that the good coding practice of validating all function return values, as well as the error handling schemes and other error conditions specific to the context. The Listing 5 shows how an error condition requires the freeing of memory and the communication of an error status to the caller.

```
int syscall32_setup_pages(struct linux_b
int npages = (VSYSCALL32_END - VSYSCALL32_BASE) >>
    PAGE_SHIFT;
struct vm_area_struct *vma;
struct mm_struct *mm = current->mm;
+ int ret;
  vma = kmem_cache_alloc(vm_area_cachep, SLAB_KERNEL
      );
  if (!vma)
@@ -78,7 +79,11 @@ int syscall32_setup_pages(struct
    linux_b
  vma->vm_mm = mm;
  down_write(&mm->mmap_sem);
- insert_vm_struct(mm, vma);
+ if ((ret = insert_vm_struct(mm, vma))) {
+   up_write(&mm->mmap_sem);
+   kmem_cache_free(vm_area_cachep, vma);
+   return ret;
+ }
  mm->total_vm += npages;
  up_write(&mm->mmap_sem);
  return 0;
```

Listing 5. Example of Improved Error Handling for CVE – 2005 – 2617

F. Zeroing Memory

Some sensitive information can be left in the memory from previous kernel operations, allowing possible private data leaks. A maintainer should preventively ensure that memory contents are wiped before using in the context where it can be read by a user process, normally by filling the memory segment with zero values. The Listing 6 shows how a disclosure vulnerability was solved by a few calls to `memset`.

G. Freeing Resources

Some security vulnerabilities can occur due to memory leaks or unreleased resources such as timers. A maintainer must ensure that good programming practices regarding resource management are enforced. Listing 7 shows how a denial of service condition was remedied by removing memory leaks.

```
    goto fail;
  }
  kaddr = kmap_atomic(page, KM_USER0);
+ memset(kaddr, 0, chunk_size);
  de = (struct ext2_dir_entry_2 *)kaddr;
  de->name_len = 1;
  de->rec_len = cpu_to_le16(EXT2_DIR_REC_LEN(1));
```

Listing 6. Example of Zeroing Memory to Solve Data Leaks in CVE – 2004 – 0685

```
     kenter("{%d}", key->serial);

     key_put(rka->target_key);
+    kfree(rka);

 } /* end request_key_auth_destroy() */
```

Listing 7. Example of Remediation of Memory Leaks for CVE – 2005 – 3119

H. Input Validation

Input validation consists of validating or modifying input data in order to ensure it complies with a safety policy by either truncation or filtering. A maintainer should ensure that all data going through a security boundary, notably all data coming directly from a user, is validated for correctness. The Listing 8 shows how input validation capabilities were not fully used for a bridge forwarding functions, allowing a vulnerability, and how it was simply solved by ensuring a specific subset of traffic for filtering.

```
  struct net_bridge *br = p->br;
  unsigned char *buf;

+ /* insert into forwarding database after filtering
     to avoid spoofing */
+ br_fdb_update(p->br, p, eth_hdr(skb)->h_source);
+
  /* need at least the 802 and STP headers */
  if (!pskb_may_pull(skb, sizeof(header)+1) ||
      memcmp(skb->data, header, sizeof(header)))
```

Listing 8. Example of Improved Input Validation Remediating CVE – 2005 – 3272

I. Capability Validation

In some cases, it is necessary to validate the executing principal's rights in an access control matrix. This is a special case of precondition validation which we considered worthy to mention separately. The Listing 9 shows how a simple check remedies a security vulnerability.

J. Fail-Safe Default Initialization

Relying on default initialization performed by the compiler can be tricky at best, especially when a simple compile option can disable the feature overall (see for example the `-Wuninitialized` and `-Wmissing-field-initializers` options in GCC[7]). Furthermore, the default initialization may not be to a value that we need for our context. As such, maintainers should ensure that critical variables are assigned to known safe

```
do_kdgkb_ioctl(int cmd, struct kbsentry
  int i, j, k;
  int ret;

+ if (!capable(CAP_SYS_TTY_CONFIG))
+   return -EPERM;
+
  kbs = kmalloc(sizeof(*kbs), GFP_KERNEL);
  if (!kbs) {
    ret = -ENOMEM;
```

Listing 9. Example of Capability Validation Solving CVE – 2005 – 3257

values, or that strings are guaranteed to be zero-terminated. Listing 10 shows how setting a pointer to NULL remedied a denial of service vulnerability.

```
  size_t array_size;

  /* set an arbitrary limit to prevent arithmetic
      overflow */
- if (size > MAX_DIRECTIO_SIZE)
+ if (size > MAX_DIRECTIO_SIZE) {
+   *pages = NULL;
    return -EFBIG;
+ }

  page_count = (user_addr + size + PAGE_SIZE - 1) >>
       PAGE_SHIFT;
  page_count -= user_addr >> PAGE_SHIFT;
```

Listing 10. Example of Fail-Safe Default Initialization Remediating CVE – 2005 – 0207

K. Protection Domain Enforcement

In the case of the Linux kernel, the data operated on can be either in user space or in kernel space. The kernel implements this differentiation by copying data between kernel and user space as needed, hereby ensuring that the information is not altered during an operation. Listing 11 shows an example of this, where the call `copy_to_user` was added with error handling.

L. Redesign

The option to redesign often requires major code changes, as functions are added or removed (or have their signatures changed), that new options appear or disappear, etc. A maintainer can improve the security of a system by changing a weak module by another which is safer (and adjusting interfaces if necessary), remove some dangerous options, etc. The Listing 12 shows how a dangerous option needed to be removed from the code.

M. Other

Other methods were not encountered in a frequency allowing us to classify them into category for themselves. Thus, the "Other" category of improvements. The Listing 13 shows a vulnerability solved by changing default file permissions.

```
case VIDIOCSWIN:
{
-  struct video_window *vw = (struct video_window *)
     arg;
-  DBG("VIDIOCSWIN %d x %d\n", vw->width, vw->height
    );
+  struct video_window vw;

-  if ( vw->width != 320 || vw->height != 240 )
+  if (copy_from_user(&vw, arg, sizeof(vw)))
+    {
       retval = -EFAULT;
+      break;
+    }
+
+  DBG("VIDIOCSWIN %d x %d\n", vw->width, vw->height
    );

+  if ( vw.width != 320 || vw.height != 240 )
+    retval = -EFAULT;
     break;
   }
```

Listing 11. Example of Protection Domain Enforcement Remediating CVE – 2004 – 0075

```
- sctp_lock_sock(sk);
-
- switch (optname) {
- case SCTP_SOCKOPT_DEBUG_NAME:
- /* BUG! we don't ever seem to free this memory. --
    jgrimm */
- if (NULL == (tmp = kmalloc(optlen + 1, GFP_KERNEL)
    )) {
-   retval = -ENOMEM;
-   goto out_unlock;
- }
-
- if (copy_from_user(tmp, optval, optlen)){
-   retval = -EFAULT;
-   goto out_unlock;
- }
- tmp[optlen] = '\000';
- sctp_sk(sk)->ep->debug_name = tmp;
- break;
+ sctp_lock_sock(sk);

+ switch (optname) {
  case SCTP_SOCKOPT_BINDX_ADD:
```

Listing 12. Example of Redesign by Removal of Option for CVE – 2004 – 2013

```
 MODULE_PARM_DESC(debug, "Enable debug output");

 module_param_named(cards_limit, drm_cards_limit,
     int, 0444);
-module_param_named(debug, drm_debug, int, 0666);
+module_param_named(debug, drm_debug, int, 0600);

 drm_head_t **drm_heads;
 struct drm_sysfs_class *drm_class;
```

Listing 13. Example of Other Solution: Changing Default File Permissions for CVE – 2005 – 3179

VI. Secure System Software Coding Guidelines

We are going to use the knowledge acquired in this study in order to summarize and improve existing secure coding checklists and guidelines [1], [14], [30], [17], [26], [27], [29], [33], [20] and apply them to kernel and system software development. Although those guidelines will overlap greatly with existing guidelines and common programmer knowledge, this contribution remains the first set of general guidelines for Linux kernel development specifically.

A. System Software Design Guidelines

A wide spectrum of errors is classified as design errors. The solutions here range over different aspects of validation, error handling, and redesign. Therefore, it is difficult to provide a checklist in this category other than suggest more formal approach to the system software development process with requirements, design, and extensive unit and acceptance testing versus just mere code hacking. Every module shall have its security requirements listed and documented and every patch to that module has to be always checked against those requirements prior application. A good example of a *failure* to do so is a replacement of a broken `cryptoloop` implementation with another, similarly broken package (see CVE-2004-2135 and CVE-2004-2136). Additionally, the coding conventions that exist as well as mandatory comments for non-trivial pieces of code should be enforced. Linus Torvalds provided such guidelines for Linux [30], but there still a lot of code that disobey them or is undocumented. It would be advisable for the developer community to gradually and systematically re-read and re-document each module/unit as it is being maintained combined with automated documentation generators (such as Doxygen [31]).

Finally, it is beneficial to maintain unit tests for all modules. One typical organization is having a separate tree hierarchy mimicking the original source code tree to include the unit tests. Another organization is to have the tests in the same hierarchy, but using a naming convention allowing to easily distinguish the tests from the implementation. Integration and regression testing helps the system at large to avoid situation where a small change in one module triggers a large failure somewhere else while not surfacing right away to the developers.

B. General Code Guidelines

As we have seen, the vast majority of errors simply come from the lack of validation of input, parameters, and return values that go in and out to functions through parameters or environment. Thus, at all cases an error handling of return values from system calls and alike must be performed (see Listing 5), a check and sanitization of all input arguments (see Listing 8) and environment variables for validity prior use, proper initialization to trusted values (such as in Listing 10) and so on. These guidelines seem obvious, but surprising even at the kernel level are not always followed, where it is a must.

Next, we need to consider termination conditions (e.g. loop termination) as well as code reachability. In some cases, loop termination conditions need to be added to ensure that a loop terminates after a finite number of iterations. One form of never-ending loops is seen in infinite recursion, which should be guarded against. Finally, we need to ensure that all paths of the code are reachable (something that can be ignored, as show in Listing 13).

Further, there are many boundary errors related to signness or data type size errors. Expected signness should always be documented for every function or variable and conversion macros to check and convert types from signed and unsigned should be used to avoid or detect integer overflows and the like. Unsigned variables should be used instead of signed ones whenever possible.

Additionally, unit testing can catch a lot of such errors, and static analysis tools [9], [10], [25] can be an useful complements. Ultimately, one of the most useful tools remains self and peer review.

C. Privileged Code Guidelines

The privileged code should be kept as short as possible [20]. Such code typically raises privileges to perform certain task (e.g. I/O) and then has to drop the privileges right after according to the least privilege principle. In the kernel, such code paths are typically in the device drivers. The danger here is that often the privileged code executes on behalf of untrusted user-level application (e.g. via system calls to read/write files or spawn processes), and as such has access to many system resources the application would not. Thus, if an application could circumvent the system code while in privileged mode, it could gain unauthorized access. Therefore, the time window where the privileges are raised, should be as short as possible.

Furthermore, it is essential that the data operated on be a copy of the data supplied by the user, and not a reference to it, as this could allow a malicious user to transform it during the processing (as illustrated in Listing 11). Finally, privileged operations need to filter input in order to avoid exploitation and increase of privilege through tainted variables.

D. Concurrent/Parallel Code Guidelines

System software, such as an operating system, should always be developed with concurrency in mind. The improper coding typically results in race conditions. The race conditions are possible where there is a more than one execution context (of a process or a thread) is attempting to access a shared resource (e.g. device, variable or a data structure, as illustrated by Listing 1). Thus, whenever dealing with global/static variables or states, acquiring proper locks or semaphores prior entry to a critical section of code and releasing them right after leaving it should be maintained. The use of static variables in functions should be avoided (or at least carefully analyzed) and local variables referring to a lockable data structure or part thereof should be initialized only after the lock has been obtained (see Listing 4). The critical section code in the code, just like privileged section code, should be as small as possible

to avoid lock contention and slowdown (due to serialization of concurrency). The critical section code must be small and thoroughly documented with expected execution scenarios to increase confidence that it is deadlock- and starvation-free. For this task some formal methods can be used, deadlock avoidance, detection, or preemption algorithms should be implemented. The latter, however, typically degrade system performance, but could be enabled as part of the debug mode.

Another way to improve concurrency and reduce locking overhead is to use copies of kernel data structures or make them immutable where possible throughout the code as no locking or mutexes are needed for immutable or copied data structures.

E. Performance Coding Guidelines

While coding for performance is a noble goal, it has to be justified in some cases as overly-optimized code can lower maintainability and be prone to security bugs. In general, it is not acceptable to remove precondition validation and error handling checks for the sake of saving a few CPU cycles.

It is important to remember that many performance issues are better resolved through profiling and proper design. For example, on single-threaded uniprocessor machines, spinlocks relying on busy-waiting should be avoided in favor of sleep-waiting primitives.

In some cases, performance issues related to long recursions become security issues (denial of service). Recursions should be replaced by bounded iterative solutions. Such a solution will increase performance and avoid a security issue.

F. Resource Management Code Guidelines

This category deals with resource leaks as in forgetting to free unused ones as well as privileged information leaks that migrates from kernel memory to user memory. Typically, all explicitly dynamically allocated resources (e.g. memory, timers, file descriptors, etc.) should be freed explicitly. If the allocation and deallocation happen in the same function, then allocation and deallocation statements (e.g. `kalloc()` and `kfree()` or `open()` and `close()`) shall used right away as if they were types of brackets while programming. One example can be seen in Listing 7. When this is not the case, proper documentation is a must, and conversion from raw resource management (allocation/deallocation) to more abstract resource management (i.e. using reference counts) should be considered. When code is reviewed, we need to ensure that the resources are deallocated only once in all code branches, a task particularly suited for static analysis tools.

Another type of erroneous resource management sometimes enables sensitive information leakage by the kernel (as show in Listing 6). This typically occurs when data structures are not zeroed by default.

G. Debug/Log Code Guidelines

Assertions and debug macros are good tools for in-place debugging and shall always be used extensively. They don't result in performance penalty when compiled with the debug macros and assertions disabled. Debug/logging should be useful and traceable to the code for debug and auditing purposes. The debug information should include timestamps, UIDs, PIDs, creation/destruction of processes and files.

VII. Conclusion

During our case study, we have found that the practices used by the Linux kernel development team to improve the security are the error handling, redesign, precondition validation, fail-safe default initialization, change of data types, ensuring atomicity, zeroing memory, input validation, protection domain enforcement, capability validation, freeing resources, unresolved, and other.

We also computed the relative frequencies of occurrence of each type of vulnerability solution. As our survey found, it is clear that a lot (about 40%) of vulnerabilities were remedied with error handling and precondition validation, which can be easily found and corrected in an audit review. We hope that this conclusion will aid developers of system software to implement proper methodologies and programming guidelines that will prevent many vulnerabilities in the future. The examples and guidelines we offer in this paper could also be furthered for training maintainers in detecting and properly correcting security vulnerabilities.

VIII. Acknowledgments

We would like to thank Dr. Mourad Debbabi for his guidance in assisting with this topic and this paper reaching the publication stage, Dr. Chadi Assi for his efforts in this work as the initial project topic in his Operating Systems Security class and the Linux Kernel Hackers who collaborated with us: Simon 'Horms' Horman, Eric W. Biederman, Nikos Ntarmos, Linus Torvalds, Andrew Morton, Greg K.-H.

References

[1] AusCERT. Secure unix programming checklist, 1996. http://www.auscert.org.au/render.html?it=1975.

[2] BitKeeper. Bitkeeper, 2005. http://linux.bkbits.net/.

[3] Bugtraq. Bugtraq, 2005. http://groups.google.ca/group/mailing.unix.bugtraq and http://www.securityfocus.com/archive/1.

[4] CERT. Cert advisories, 2005. http://www.us-cert.gov/cas/techalerts/index.html.

[5] Shuo Chen, Zbigniew Kalbarczyk, Jun Xu, Ravishankar, and K. Iyer. A data-driven finite state machine model for analyzing security vulnerabilities. In *2003 International Conference on Dependable Systems and Networks (DSN'03)*, page 605. IEEE, 2003.

[6] Various Contributors. *Linux Kernel Mailing List Archives via Google*. 2005. http://groups.google.com/groups?q=linux+kernel.

[7] Various Contributors and GNU Project. *GNU Compiler Collection (GCC)*. Free Software Foundation, Inc., 1988-2005. http://gcc.gnu.org/onlinedocs/gcc/.

[8] Debian. The debian's svn repository, 2005. http://svn.debian.org/wsvn/kernel/.

[9] Dawson Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of OSDI 2000*. Usenix, 2000.

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

[11] Steve Hamm. Linux inc. *BusinessWeek Online*, 01 2001. http://www.businessweek.com/magazine/content/05_05/b3918001_mz001.htm.

[12] Brian Hatch, James Lee, and George Kurtz. *Hacking Exposed Linux, 2nd Edition*. McGraw-Hill Osborne Media, 2002.

[13] Simon Horman. Ultra monkey: Kernel security bug database, 2005. http://www.ultramonkey.org/bugs/cve/.

[14] M. Howard and D. LeBlanc. *Writing Secure Code, 2nd edition*. Microsoft Press, 2002.
[15] Xie Huagang. Lids: Linux intrusion detection system. http://www.lids.org/.
[16] Jack Koziol, David Litchfield, Dave Aitel, Chris Anley, Sinan "noir" Eren, Neel Mehta, and Riley Hassell. *The Shellcoder's Handbook : Discovering and Exploiting Security Holes*. John Wiley & Sons, 2004.
[17] Macadamian. Macadamian's code review checklist. http://www.macadamian.com/index.php?option=com_content&task=view&id=27&Itemid=31.
[18] D. Mackenzie, P. Eggert, and R. Stallman. *Comparing and Merging Files*. Free Software Foundation, 2002. http://www.gnu.org/software/diffutils/manual/ps/diff.ps.gz.
[19] Scott Mann, Ellen Mitchell, and Mitchell Krell. *Linux System Security*. Pearson Education, 2002.
[20] Sun Microsystems. Security code guidelines, 2000. http://java.sun.com/security/seccodeguide.html.
[21] NIST. National vulnerability database, 2005. http://nvd.nist.gov/.
[22] NIST. National vulnerability database statistics, 2005. http://nvd.nist.gov/statistics.cfm.
[23] OpenWall. Openwall gnu/*/linux (owl) - a security-enhanced server platform. http://www.openwall.com/Owl/.
[24] Marc-André Laverdiére Papineau. Towards Systematic Software Security Hardening. Master's thesis, Concordia Institute for Information Systems Engineering, Concordia University, August 2007.
[25] B. Schwarz, Hao Chen, D. Wagner, J. Lin, Wei Tu, G. Morrison, and J. West. Model checking an entire linux distribution for security violations. In *Proceedings of the 21st Annual Computer Security Applications Conference*, pages 13–22. IEEE, 2005.
[26] R. Seacord. *Secure Coding in C and C++*. SEI Series. Addison-Wesley, 2005.
[27] Adam Shostack. Security code review guidelines, 2004. http://www.homeport.org/~adam/review.html.
[28] Stephen Smalley, Chris Vance, and Wayne Salamon. Implementing selinux as a linux security module. http://www.nsa.gov/selinux/papers/module.pdf.
[29] Visual Studio Team System. Guidelines for writing secure code. http://msdn2.microsoft.com/en-us/library/ms182020.aspx.
[30] Linus Torvalds. Linux kernel coding style. http://www.llnl.gov/linux/slurm/coding_style.pdf.
[31] Dimitri van Heesch. *doxygen Manual for version 1.4.6*. Doxygen, 2004. ftp://ftp.stack.nl/pub/users/dimitri/doxygen_manual-1.4.6.pdf.zip.
[32] Steven J. Vaughan-Nichols. Linux server market share keeps growing. *Linux-Watch Online*, May 2007. http://www.linux-watch.com/news/NS5369154346.html.
[33] D. Wheeler. Unix and linux secure coding howto, 2003.
[34] R. Wita and Y. Teng-Amnuay. Vulnerability profile for linux. In *Proceedings of the 19th International Conference on Advanced Information Networking and Applications*, pages 953–958. IEEE, 2005.
[35] C. Wright, C. Cowan, J. Morris, S. Smalley, and G. Kroah-hartman. Linux security modules: General security support for the linux kernel, 2002.
[36] Chris Wright. *OOPS Linux Kernel Security Patches in the GIT Repository*. 2005. http://www.kernel.org/git/?p=linux/kernel/git/chrisw/stable-queue.git;a=tree.