Defensive Programming for Red Hat Enterprise Linux (and What To Do If Something Goes Wrong)

Ulrich Drepper
Red Hat, Inc.
drepper@redhat.com

April 8, 2009

Abstract

Programming Language APIs and their libraries are often not designed with safety in mind. The API of the C language and the Unix API (which is defined using C) are especially weak in this respect. It is not necessarily the case that all programs developed using these programming languages and APIs are unsafe. In this document we will describe possible problems, how to prevent them, and how to discover them.

1 Introduction

The Internet used to be a nice place where "friendly" people met. When the first Denial of Service (DoS) event happened this was more a novelty than a problem. Today security is a major issues since the stakes are higher. There is real money on the line and there are criminals trying to get their hands on it. In addition, cracking machines is a "sport" for some degenerates that might result indirectly to financial losses due to downtime and system administrator overtime.

The programming environments developed for today's operating systems are not designed with security in mind. The main focus of the C and Unix APIs was efficiency in performance and memory-usage. This heritage has still repercussions today since C is still one of the major programming languages in use, and the Unix API still has to be used even with other programming languages which do not have the security-related disadvantages of C.

There is no reason, though, that programs developed with these APIs have to be insecure. It just means that some more effort is necessary to write code which can be securely deployed. This paper covers several aspects of the C and UNIX APIs that need attention and it covers the aspects of a program that need to be handled to ensure security. This paper does not completely cover the topic, but it provides insight into the problems most often exploited in today's programs.

The main aspects of security covered here are:

- Reducing[1] bugs in programs;
- Restricting the effects of bugs in programs;
- Restricting the effects of user errors;
- Proper coding practices– including recommendations on which functions to avoid.

[1] "Eliminating" is really too much to expect.

Authentication and authorization are pretty much independent from the first three points and authorization and authentication are demanded by the program semantics. Without this requirement, adding authentication and authorization is not really useful since the semantics of the program changes.

Problems are avoided and recognized in a number of phases: during programming, compilation, runtime, and debugging. We will describe the possibilities in all these phases.

2 Safe Programming

The main problem with programming in C, C++, and similar languages is the memory handling. Memory for most interfaces has to be allocated explicitly and this means possibilities for bugs. These memory handling problems are pervasive and in the last few years have become the main reason for exploits. A large array of techniques has been developed by the black hat groups to exploit memory handling bugs. These bugs mainly include buffer overruns, double `free()` calls, and invalid pointer value usage. In later sections we will discuss how to detect these kind of bugs. Here we concentrate on ways to avoid them altogether.

Other topics in this section include writing code that minimizes the effects that an exploit of a program bug can have. Giving an intruder access to the whole system is much worse than having the process die. Mitigating the risks is important but unfortunately not often performed. This will also help to some extent with problems which

happen due to bugs or incorrect program designs without the influence of an intruder. The result could be that system resources are exhausted and system performance is negatively impacted.

2.1 Counter the C Memory Handling Problems

The C language has no support for memory allocation in the language itself. The standard library contains the `malloc()` family of functions but it has so many possibilities for incorrect use, that memory allocation is always a weak point. C++ shares the problems to some extent, but the `new` operator and the standardization of higher-level data structure helps to reduce the direct use of the `malloc()` functions.

Direct use of `malloc()` and use of C has its advantages: the programs have the possibility to be measurably more efficient. But the required skill level to do it right and efficiently is high and errors slip in easily. Therefore it is desirable to eliminate explicit memory allocation. If memory must be explicitly allocated, it is necessary to keep track of the size of the allocated memory blocks and make sure the boundaries are respected by all the code touching the memory.

2.1.1 Respecting Memory Bounds

When writing into a memory block, the program (and programmer) must at all times be aware of the size of the memory block. If a function gets passed a reference to a memory block, the size of the block needs to be known to the function as well. This can happen implicitly: if the reference is to a single object of a given type, providing only the reference with the correct type is enough. Unless the types of memory blocks are not maintained correctly, this is sufficient. It is necessary, though, to remember type conversions and, if they are necessary, do them correctly. The compiler takes care of the rest by enforcing correct use of the type.

There are a few traps. First, all too often a simple mistake in the use of `malloc()` causes havoc:

```
struct example *ptr =
  (struct example *) malloc(sizeof(ptr));
```

The allocated memory is in general not sufficient for an object of the given type `struct example` since the parameter for `malloc()` should be `sizeof(*ptr)` instead. A big difference. In this situation it might be useful to use a macro like this:

```
#define alloc(type) \
  (type) malloc(sizeof(*(type)))
```

This macro makes sure the dereference in the `sizeof` operator is not forgotten and the compiler makes sure that incorrect uses of the type are recognized. The type system can be defeated, though, when pointer parameters or variables are of type `void *`. For this reason it is almost vital to use `void *` only when really necessary. If a variable or function parameter/return value needs to be used for more than one pointer type, a `union` should be used. The additional syntax needed to express the necessary assignments do not translate into runtime costs so it has no disadvantages.

If an array of objects is needed, the size of the array should always be passed to a function which gets a reference to the array passed. This way the implementation of the called function can always check whether the array boundaries are violated or not. It is of advantage to use the a GNU C compiler to express this:

```
int addvec(int n, int arr[n]);
```

This kind of parameter list allows the compiler to check at the call site whether the passed array is big enough. In the function definition the compiler could check accesses to the array. It should be noted that the GNU C compiler does not currently check for this. Future versions of GNU C will hopefully change this; the more code uses the annotation, the more the compiler will be able to check when it becomes available. Note that existing interfaces can be converted to the extended syntax even if the size parameter follows the array parameter. The example above would need to be written as follows:

```
int addvec(int n; int arr[n], int n);
```

This is a rather strange syntax. The part before the semicolon in the parameter list is a forward declaration of the parameters, which allows to use parameters later in the list.

An unfortunate fact is that the standard runtime library contains interfaces which violate these rules. Two examples are these:

```
char *gets(char *s);
char *getwd(char *buf);
```

These two functions should under no circumstances ever be used. The `gets()` function reads input from standard input and stores it into the buffer pointed to by `s`. The input can in theory be arbitrarily long and could overflow the buffer. The problem with `getwd()` is similar. The path of the current directory is stored in the

buffer pointed to by `buf`. The problem is that the maximum path name length is not necessarily known at compile time, or runtime, or at all (i.e., it can be unlimited). Therefore it is possible to overflow the provided buffer and the `getwd()` implementation cannot prevent this. Instead of these two functions programmers should always use `fgets()` and `getcwd()`. These alternative interfaces have almost the same interface but the length of the buffer is passed to the function as well.

The `getcwd()` interface shows the next step in the evolution of interfaces or, at least, the way interfaces *should* be implemented. As an extension of the GNU C library, the buffer parameter passed to `getcwd()` can be `NULL`. In this case the `getcwd()` implementation allocates a buffer large enough to hold the return value as if it were allocated with `malloc()`. This ensures the buffer is never too small. The drawback is that there is a overhead at runtime: the buffer has to be allocated and freed at runtime. This overhead is usually not measurable. The `malloc()` implementation in the GNU C library is fast enough for this.

2.1.2 Implicit Memory Allocation

If interfaces are newly developed it is advisable to design them so that necessary buffers are allocated in the interfaces themselves. There are several extensions to standard interfaces and some new interfaces in the GNU C library which take over the memory allocation. These interfaces should be used whenever possible.

Other functions operating on file names as `getcwd()` have been similarly extended. `realpath()` is one such example. This function should be avoided, when possible, and instead the following interface should be used:

```
char *canonicalize_file_name(const char *)
```

The effect is the same as `realpath()`'s, but the result is always returned in a newly allocated buffer.

The next interface in this group helps to avoid problems with `sprintf()` and `snprintf()`. These interfaces help creating strings using the sophisticated functionality of the `printf()` function family. The problem is the passed-in buffer must be sized appropriately. If no absolute, never-exceeded limit is known, `sprintf()` must never be used since it provides no support to prevent buffer overflows. To use `snprintf()` in such a situation it is necessary to call `snprintf()` in a loop where the function is called with ever larger buffers. The GNU C library provides two more functions in this family:

```
int asprintf(char **strp, const char *fmt,
             ...);
int vasprintf(char **strp, const char *fmt,
              va_list ap);
```

These interfaces work just like the rest of the `sprintf()` functions but the buffer, which contains the result, is allocated by the function and therefore is never too small. The caller's responsibility is only to free the buffer after it is not needed anymore. Even though the memory allocation has to be performed at each call, the use of these interfaces might in fact be faster since it is not necessary to perform possibly complicated computations to determine the buffer size in advance or call the `snprintf()` function more than once in a loop in case the buffer proved to be too small.

There are some limits on how much can be done with `asprintf()`, though. All the output has to be performed using one format string and the set of parameters passed to the function must be passed to the function at compile time. It is not possible to dynamically construct the parameter list at runtime. For this reason the GNU C library contains another new interface that allows the construction of arbitrarily complex strings:

```
FILE *open_memstream (char **strp,
                      size_t *sizep);
```

As the signature of the interface suggests, this function creates a new `FILE` object which subsequently can be used with all the functions which write to a stream. The parameters passed to the interface are pointers to two variables which will contain the address and the length of the final string. The actual values will not be filled in until the string is finalized by a call to `fclose()`. This means the `open_memstream()` call has no effect on the content of these variables nor is it possible to track the constructions of the string this way.

If an existing string, or parts thereof, simply has to be duplicated it is not necessary to use any of the interfaces; this would be overkill. Instead the GNU C library provides four interfaces which do just this job and nothing else and are therefore more efficient; there is no possibility of computing the memory buffer size incorrectly:

```
char *strdup(const char *s);
char *strndup(const char *s, size_t n);
char *strdupa(const char *s);
char *strndupa(const char *s, size_t n);
```

`strdup()` is the simplest function of this group. The string passed as the one parameter is duplicated entirely in a newly allocated block of memory which is returned. The caller is responsible for freeing the memory, which

means the lifetime of the string is not limited. `strndup()` is similar, but it duplicates at most only as many bytes as specified in the second parameter. The resulting string is always NUL-terminated.

The `strdupa()` and `strndupa()` interfaces are similar to the already mentioned two interfaces. The only difference is that the memory allocated using these latter two interfaces is allocated using `alloca()` and not `malloc()` (which also implies that these two interface are implemented as macros and not real functions). Memory allocation using `alloca()` is much faster but the lifetime of the string is limited to the stack frame of the function in which the allocation happened and it is not possible to free the memory without leaving the frame. If variable sized arrays are used in a function, using `alloca()` as well causes problems since the compiler might not be able to deallocate the memory for the variable sized array even if control left the block it is defined in. The reason is that the memory blocks returned by all `alloca()` calls made after entering the block with the array would also be freed. Often these limitations are no problem and then using `strdupa()` and `strndupa()` has its advantages.

A last group of interfaces which operate on strings and process possibly unlimited-length input are the `scanf()` functions. Although these functions are rarely usable in robust code (they are in the author's opinion a complete design failure) some programmers choose to use them and consequently run into problems. The issues here are the `%s` and `%[` format specifiers. These two specifiers allow the reading of strings which might come from uncontrolled sources and therefore have the potential of being arbitrarily long. It is possible to use the precision notation to limit the number of bytes written to the string:

```
int num;
char buf[100];
scanf("%d %.*s", &num,
      (int) sizeof(buf), buf);
```

It is even possible, as in this example, to dynamically determine the buffer size and pass the information to the `scanf()` function. But all this pre-computation of the buffer size is just as error-prone as int the other cases. This is why the GNU C library has another extension which does away with these problems. The above example can be rewritten as this:

```
int num;
char *str;
scanf("%d %as", &num, &str);
```

The `'a'` modifier in this format string indicates that the `scanf()` function is supposed to allocate the memory needed for the parsed string. A pointer to the newly allocated memory is stored in the variable `str`. Once again, the size of the allocated block is never too small and the program is becoming simpler. So, if it is deemed necessary to use the `scanf()` functions to parse input, using the `'a'` modifier is advisable. Readers who know the ISO C99 standard will notice that this standard defines `'a'` as a format (for parsing floating-point numbers in hexa-decimal notation). The GNU C library supports this as well. The implementation of these functions will try to determine from the context in the format string what the caller means.

More often than not, the formatted input using `scanf()` is not desirable since these interfaces make gracious error handling hard. Often entire lines are first read into memory and then processed with hand-crafted code. In the previous section it was said that `fgets()` should be used instead of `gets()`. This is true, but not the final word. `fgets()` requires the caller to allocate the buffer. The GNU C library contains interfaces, which implicitly allocate memory:

```
ssize_t getdelim(char **lineptr, size_t *n,
                 int delim, FILE *stream);
ssize_t getline(char **lineptr, size_t *n,
                FILE *stream);
```

`getdelim()` reads from the stream passed as the final parameter until a character equal to `delim` is read. All the resulting text is returned in the buffer pointed to by `*lineptr`. This buffer is enlarged, if necessary; the caller does not have to worry about it. The prerequisite for using this interface is that `*lineptr` initially must point to a buffer allocated with `malloc()` or must point to a `NULL` pointer. The value pointed to by `*n` must be the size of the buffer (zero in case of a `NULL` pointer). The `getline()` function is implemented using `getdelim()` function, with `'\n'` as the value for `delim`. It is not necessary to allocate a new buffer for every new call. Instead, the code using this function usually looks like this:

```
char *line = NULL;
size_t linelen = 0;
while (!feof(fp)) {
  ssize_t n = getline(&line, &linelen, fp);
  if (n < 0)
    break;
  if (n > 0) {
    ... process line ...
  }
}
free(line);
```

This code will never have problems with buffer overruns due to overly long lines. The buffer is reused from one

getline() call to the other and thusly the performance penalties resulting from the implicit memory allocation is minimized. Additional advantages over the fgets() function are that both functions can handle NUL bytes in the input and that getdelim() can use any record separator, not just newline.

2.2 Defeating Filesystem-Based Attacks

Completely unrelated to memory handling problems, a second big group of security problems in programs is related to the interaction with filesystems. Making sure that a program only modifies the files it is supposed to modify can be hard. The problems described in this section are not limited to code written in C. On the contrary, all programming languages use the same set of interfaces in their bindings for the operating system and therefore all that is said in this section is valid for all programming languages. A problem could be that not all language runtimes provide access to all the functions mentioned here.

The Unix semantics of a file distinguishes between the content of the file itself and the binding of the content to one or more names in filesystems. The content of a file can exist even if no name is bound to it; this requires an open file descriptor. When the file descriptor is closed, the content of the file is removed. These semantics proved to be powerful and extremely useful. The problem is that programmers must at all times be aware of these semantics and be aware that at any time the relationship between file names and file content can change. All that is needed is write access to the filesystem.

2.2.1 Identification When Opening

Robust programs will not simply trust file names. They will try to identify files before they are used. Identification is, for instance, possible by comparing file ownership, creation time, or even location of the data on the storage media. This information is available through the stat() system calls. Somebody might therefore write code like this:

```
stat(filename, &st);
if (S_ISREG(st.st_mode)
    && st.st_ino == ino
    && st.st_dev == dev) {
  fd = open(filename, O_RDWR);
  // Use the file
  ...
  close(fd);
}
```

The information stored by stat() in st is used in the if statement to identify the file. Only when these tests are successful is the file opened and used. This looks safe but there are problems. There is no guarantee that the name in filename refers to the same file in the stat() and open() call. Somebody might have changed this. This might seem unlikely, but an intruder will use exactly these problems by maximizing the probability. As a result the test performed before the open() call can be invalidated by the time the kernel tries to open the file. Correctly written, the above example would look like this:

```
fd = open(filename, O_RDWR);
if (fd != -1 && fstat (fd, &st) != -1
    && st.st_ino == ino
    && st.st_dev == dev) {
  ...
}
close (fd);
```

In this revised code the decision whether to use the file is made after opening the file. Opening it does not yet mean making use of the file, so this is no semantical change. The improvement in the code comes from the use of fstat() instead of stat(). fstat() takes a file descriptor as the parameter, not the file name. This way we can be 100% sure that the data retrieved via fstat() is for the same file we opened. If somebody removes or replaces the filename between the open() and fstat() call this has no effect since, as explained in the introduction to this section, the file content remains accessible as long as there is a file descriptor. No operation can cause the link between the file descriptor and the file content to be broken.

The new code might performance-wise be a little bit at a disadvantage. If the possibility of a mismatch is high, the unconditional open()/close() pair has a measurable impact. If this is a problem, the program can still use stat() as in the original code. In this case it is necessary to repeat the test as in the second example with fstat(). This way one gets the best of both worlds.

There are a number of other functions which have similar problems. In all cases the one variant uses a filename for identification, in the other case a file descriptor.

file name	file descriptor
stat()	fstat()
statfs()	fstatfs()
statvfs()	fstatvfs()
chown()	fchown()
chmod()	fchmod()
truncate()	ftruncate()
utimes()	futimes()
chdir()	fchdir()
execve()	fexecve()
setxattr()	fsetxattr()
getxattr()	fgetxattr()
listxattr()	flistxattr()
getfilecon()	fgetfilecon()

The first seven interfaces provide protection in basically the same way we have seen in the previous example. Whenever there is the possibility that the filesystem has been modified and the file name might refer to a different file, then the `f*()` variants of the functions should be used.

The `chdir()` function is special since it operates on directories, not files. Directories cannot simply be removed. They need to be empty to allow this. While it is a possible attack angle, the main issue with `chdir()` is changing the current working directory using relative paths. This is a problem in the presence of symbolic links. Assume the following directory hierarchy:

```
l1/
l1/l2a/
l1/l2a/l3/
l1/l2a/l3/l4/
l1/l2b -> l2a/l3/l4
```

The only noteworthy thing is the symbolic link named `l2b`. A program, started in the directory `l1`, which then executes `chdir("l2b")` followed by `chdir("..")` does not arrive back in `l1` as one might expect. Instead it finds itself in `l1/l2a/l3`. If an attacker could create symbolic links like `l2b` in the example, this might lead to possible problems. They can be especially severe since symbolic links can cross device boundaries. I.e., the symbolic link might refer to filesystems anybody has full write access to, like in general `/tmp`. There are basically two ways to prevent this surprise:

1. By preventing the following of the symbolic link in the first `chdir()` call. The semantics of the program have to decide whether this is a valid option. If it is, the `chdir()` call should be replaced with the following sequence:

   ```
   int safe_chdir(const char *name) {
     int dfd = open(name,
                    O_RDONLY|O_DIRECTORY
                    |O_NOFOLLOW);
     if (dfd == -1)
       return -1;
     int ret = fchdir(dfd);
     close(dfd);
     return ret;
   }
   ```

   This function can be used just like `chdir()` but the operation will never follow a symbolic link, thus avoiding the problem in the example.

2. By avoiding the relative path in the second `chdir()` call. To make sure the program returns to the original directory, the following additional steps could be performed:

   - before the first `chdir()` add a new function call:

     ```
     int dfd = open(".", O_RDONLY)
     ```

     This descriptor has to be kept open.

   - instead of the problematic `chdir("..")` call one uses `fchdir(dfd)` to do the actual work followed by `close(dfd)` where the parameter in both calls is the descriptor opened in the first step.

   This sequence guarantees that the program returns to the original directory since the `open()` call is guaranteed to provide a descriptor for the current working directory in use at that time.

   Note that using `getcwd()` to retrieve the name of the current working directory and then later using the absolute path will not help either. Somebody might have changed a directory somewhere along the path. Open file descriptors are the only reliable way to switch back and forth between two directories.

There is one problem with using `fchdir()` on Linux. If the user has only execution permission and no read or write permission to the directory, the use of `fchdir()` is not possible. Having directories with this permission is not unreasonable: it allows hiding the directory structure below the protected directory:

```
$ find secret -printf '%m %p\n'
711 secret
775 secret/random
755 secret/random/binary
```

In this example `secret` is a directory which is readable and writable only by the owner. Others still can execute the binary in the subdirectory `random` but they have to know the complete path. By using non-guessable names for the directory here named `random` one can achieve some level of security. The use of `fchdir()` to change to the secret subdirectory would be impossible for anyone other than the owner. This means the safe `chdir()` replacement must be used with care.

If directories just have to be traversed the `ntfw()` function of the standard C library can be used. The third parameter this function expects is the number of file descriptors the `nftw()` is allowed to use for internal purposes. I.e., the function uses descriptors internally for the directories it traverses and so it guarantees the operation cannot be disturbed by changes to the directory tree. One possible problem appears if the directory tree is deeper than the number of descriptors allowed but this is something the programmer can control.

As for the fexecve() interface, it might not be obvious when it is useful to use it. One plausible example is code which needs to verify the binary has not been tampered with. An executable itself or a global database could contain checksums of known-to-be good executables and their names. When execution of a binary is requested, the following steps can be performed:

1. check MD5 checksum of binary
2. if it matches, run the binary

As with the stat() example above, we would expose ourselves to an attack if the checksumming and the execution would be performed independently since there would be no guarantee that the checksummed binary is the same as the binary which gets executed. fexecve() helps in the obvious way. The code which performs the checksumming opens a file descriptor for the binary, then gets the binary content to compute the checksum. If the checksum matches, the file descriptor is not closed. Instead it is used in the fexecve() call to start the execution. A safe variant of the execve() function that automatically computes the checksums could be written. The only drawback of such a userlevel implementation is that the program needs read access in addition to execution access to the file. Note there is no direct kernel support for the fexecve() function. It is implemented at userlevel using the /proc filesystem. If this filesystem is not mounted, the fexecve() function will fail.

The getxattr(), setxattr(), and listxattr() functions are the interface to the extended attribute handling for filesystem objects. The issues with using them is equivalent to that of stat(): information about a specific file might in fact set and get information about a different file if the file has been replaced. getfilecon() is part of the SELinux improvements. It can be thought of as more or less a wrapper around getxattr() and therefore requires the same care when using it.

A few more words about the O_NOFOLLOW flag mentioned in the fchdir() description above. This flag is available for all open() calls, not just for directories. It allows all file open operations to avoid following symbolic links. Symbolic links are very useful to help with filesystem layout compatibility issues: a file is moved to a new location and a symbolic link is left in the place of the old location. Therefore it is not generally a good idea to use this flag since it prevents this compatibility from working.

There are filesystem operations, though, where symbolic links should never be followed. For instance, the actual location of temporary files is not important. These are also the files most vulnerable to symlink-attacks since the directory for the temporary file (/tmp or /var/tmp) is writable for every user. An attacker could therefore create a symbolic link with a name which the attacked program will open. As a symbolic link the name can point to any file on the filesystem and thusly expose the content of every file. All that is needed is the superuser, or an SUID root binary trying to read the file through the symlink. A programmer should therefore at all times distinguish between files which have to be reused over a long time and possible several revisions of the program, and files which exist only briefly. In case of the latter the use of O_NOFOLLOW is almost always correct. For the former class, judgment on part of the developer is needed.

2.2.2 Safely Creating Files

Creating new files is another cause of problems. When not done correctly, it is easy to overwrite files unintentionally, erase previous content, make files available to others unwillingly, etc. Creating new files is especially important when creating temporary files.

For robust programs it is almost always best to not create new files with the final name. By doing this, it is easier to recover if the program is interrupted before finalizing the initialization of the newly created file. After the restart the program does not have to determine whether the file is fully initialized or not. A requirement for this kind of file creation and initialization is that the temporary file created on the same device as the final file. Only then is it possible to atomically move the temporary file in the final position. Renaming and replacing is covered in the next section.

The best general solution for creating temporary files is to use mkstemp():

```
int mkstemp(char *template);
```

The parameter must be a writable buffer which is filled with a file name whose last six characters are XXXXXX. These six characters will be replaced by the mkstemp() call with characters which make the filename unique. The caller can use the buffer after the call to determine the file name. The return value is a descriptor for the opened file. This file is guaranteed to not have existed before and is empty. The initial access permissions are 600, i.e., read and write access are granted only for the owner. If the file needs to have other permissions afterward, the program should use fchmod() to change the permissions.

If the file is really temporary and need not be preserved, the file can be unlinked right after creation. The file descriptor remains usable until being closed. If the temporary file can be created in /tmp instead of a directory determined by the program, an alternative to using mkstemp() is the following interface:

```
FILE *tmpfile (void);
```

The file associated with the returned stream is not associated with any name in a filesystem, it is anonymous from the start and cannot be linked with a name. The required storage is allocated in the `/tmp` directory. If these conditions are acceptable, using `tmpfile()` can have advantages. The biggest obstacle might be the use of `/tmp` since the directory/device might have to be shared with all the other users and therefore resources can be depleted easily.

There are a couple of other interfaces which generate names for new, non-existing files (`mktemp()`, `tempnam()`, `tmpnam()`). They do not generate new files, just the names. It is up to the program to create the file correctly without destroying any data. The key is to use the `O_EXCL` flag with `open()`. This flag ensures that existing files are not overwritten. If there is a file with the provided name, the `open()` call fails. In this case the procedure needs to be repeated, starting with the generation of a new file name. Code using `mktemp()` could look like this:

```
static const char template[]
  = "/some/dir/nameXXXXXX";
char buf[sizeof(template)];
int fd;
do {
  strcpy(buf, template);
  if (mktemp(buf) == NULL)
    fail();
  fd = open(buf, O_RDWR|O_CREAT|O_EXCL,
            0600);
} while (fd == -1);
```

There are some situations when using the functions which only generate names are useful. But these situations are rare. The general advice to only use `mkstemp()` and `tmpfile()` is good.

There is one more interface to discuss here. In some situations it is useful to create new directories in the same way new files are created. This is mainly useful when creating temporary directories.

```
char *mkdtemp(char *template);
```

The parameter is the same as it is in the case for `mktemp()` and `mkstemp()`, a pointer to a buffer with a name ending in `XXXXXX`. If possible, a uniquely named directory is created and a pointer to the name is returned. Temporary directories are more difficult to handle, though. It is not possible to unlink the directory after it has been created and keep a descriptor open. It is not possible to unlink non-empty directories and once a directory is unlinked, it is not possible to create new files. This means the cleanup of temporary directories can be tricky.

2.2.3 Correct Renaming, Replacing, and Removing

The standard runtime provides five interfaces for the implementation of renaming, replacing, and removing operations:

```
int unlink(const char *pathname);
int rmdir(const char *pathname);
int remove(const char *pathname);
int rename(const char *oldpath,
           const char *newpath);
int link(const char *oldpath,
         const char *newpath);
```

These interfaces are working well but care must be taken to avoid destroying valuable data and handle race conditions. One situation is the creation of a new, unique file in a robust way, even when multiple processes could execute the same code at the same time. A first attempt at writing this code might look like this:

```
int fd = open(finalpath, O_RDWR);
if (fd == -1
    || fstat(fd, &st) == -1
    || extratests(fd, &st) != 0) {
  fd = mkstemp(tmppath);
  // initialize file FD
  ...
  rename(tmppath, finalpath);
}
```

While this code would behave correctly when there is only one copy of the program running, it can create problems when multiple copies are in use at the same time. Since two processes could arrive at the `fstat()` call at about the same time, both could decide to create temporary files and then create the final name for the file. The problem is that the `rename()` function will mercilessly replace any existing file with the target's name. This means two or more processes might think that they successfully initialized the file and act accordingly. Worst of all, the process whose file name got overwritten will not update the surviving file. A more correct way to perform the operation is as follows:

```
int fd;
while (1) {
  fd = open(finalpath, O_RDWR);
  if (fd != -1
      && fstat(fd, &st) != -1
      && extratests(fd, &st) == 0)
    break;
  if (fd != -1) {
    // the file is not usable
```

```
    char buf[40];
    sprintf(buf, "/proc/self/fd/%d", fd);
    unlink(buf);
    close(fd);
  }
  fd = mkstemp(tmppath);
  // initialize file FD
  ...
  if (link(tmppath, finalpath) == 0) {
    unlink(tmppath);
    break;
  }
  close(fd);
  unlink(tmppath);
}
```

This code looks much more complicated but this complication is necessary. For instance, after recognizing an invalid file, one cannot simply use `unlink(finalpath)` since by the time this all is performed, the file with this name might already be a different one than the one associated with `fd`. This is why the `/proc` filesystem has to be used this way. The actual creation of the file with the final name is not performed using `rename()`; this would be unreliable.

The problem of just reliably renaming a file is similarly complicated. If a `rename()` call is not sufficient because you have to guarantee that the file being renamed really is the one intended, something more complicated is needed. Once again you have the disconnect between file descriptors (which allow identifying a file) and file names.

```
int frename(int fd, const char *newname)
{
  char buf[40];
  sprintf(buf, "/proc/self/fd/%d", fd);
  size_t nbuf2 = 200;
  char *buf2 = alloca(nbuf2);
  int n;
  while ((n = readlink(buf, buf2, nbuf2))
         == nbuf2)
    buf2 = alloca(nbuf2 *= 2);
  buf2[n] = '\0';
  static const char deleted[]
    = " (deleted)";
  if (n < sizeof(deleted)
      || memcmp(buf2 + n
                - (sizeof(deleted) - 1),
                deleted,
                sizeof(deleted) - 1) != 0)
    return rename(buf2, newname);
  errno = ENOENT;
  return -1;
}
```

This code relies, once again, in the `/proc` filesystem. The files in `/proc/self/fd` are actually symbolic links and if they refer to a file which has in the meantime been deleted, this is indicated in the symbolic link content. This code has one limitation: if the real file name ends in `" (deleted)"`, the function will always fail. In reality this should not be too problematic since these names can easily be avoided.

It is usually necessary to go to the extra length of creating a temporary file and then rename it since otherwise there is a point in time when there is no file with the name in question. The `rename()` function is atomic and guarantees that at all times an `open()` call will succeed.

2.3 Reducing the Risk

The next step after fixing immediate bugs and problems in a program is to reduce the effects of an exploit of remaining bugs. The goal is to minimize the amount of privileges a program has at any time. The privileges a program has include:

- Memory access protection. Memory mapping can be either no, read, write, or read/write access. In addition there are usually execution permissions. There are some problems with the latter on the IA-32 architecture, but some techniques such as Red Hat's Exec-Shield [2] help to overcome the problem.
- File access permissions. Different permissions are granted to user, group, and all others.
- Process user and group ID. Each process has an effective user and group ID which determine whether certain operations are allowed or not.
- Filesystem user and group ID. The Linux kernel provides separate permissions which allow controlling the filesystem operations a process is allowed to do.
- File descriptor operations allowed. Each descriptor is created with a mode which allows read, write, or both operations.
- Resource allocation permissions. System resources have limits which a process cannot exceed.

2.3.1 Memory Access

The most fundamental permissions are those controlled by the processor. Modern processors and OSes allow memory pages to have individual permissions for read, write, and execution. The ideal situation is that pages with program code have read and execution permission, read-only data pages have only read permission, and data pages and the stack have read/write permission. This way neither program code nor read-only data can be modified, and no data can be executed.

The compiler and linker try to arrange the program code and data parts of the program as well as possible. The

stack's permissions are controlled by the OS. Extensions in Red Hat Enterprise Linux allow you to restrict the permissions for the stack so that whenever possible they do not include execution permission. Refer to [2] for more information. The programmer's responsibility is to define as much data as possible as constant. [1] explains this in detail (everything applies to executables as well).

What remains is to correctly allocate memory at runtime. The `malloc()` interface is not an issue, the memory allocated with it has the permissions determined by the OS. In Red Hat Enterprise Linux the memory has only read and write permission.

The important interfaces are `mmap()` and `mprotect()`. Memory allocated with `mmap()` can have any of the three permissions set, individually selectable for each page. `mprotect()` enables you to change the permissions after the initial mapping.

The rule of the least possible permissions requires that pages that contain only code are not mapped with write access, and that pages with changeable data do not have executable permissions set. This has some consequences for program design. For instance, it might be efficient to map a DSO in a single piece with read, write, and execution permissions–only one mmap call is necessary and everything is allowed. However, this is insecure. Therefore, all files that are meant to be mapped into memory should be segmented: for each part of the file, determine whether it contains code, read-only data, or changeable data. The parts of the file should be grouped according to these criteria so that there are at most three segments.

Since protection can only be selected with page granularity there are two ways to go ahead:

- align the segments in the file at page boundaries. This is wasting on average half a page on disk. This might be a problem if the page size one has to account for is very big.
- do not use alignment in the file, but map the pages at the boundaries of the segments twice, one time as the last page of one segment, a second time as the first page of another segment. The issues with this are that on average one page of memory per segment boundary is wasted, and that the file construction is more complicated if there is a dependency between the location of two segments.

This sounds all quite complicated but it is necessary. Allocating memory with too much permission is very dangerous and should be avoided under all circumstances.

The programmer can do even better than this in some situations. Some parts of the files might have to be writable at some time. If after an initialization phase the values are not modified anymore, the program should use `mprotect()` to remove write access. In some special circumstances it might even be useful to change the permission to forbid writing even if the data has to be written to again later. This requires that before every write access the permission is changed. For high-risk data the high performance cost of switching permissions back and forth is justifiable.

Changing the permissions is also desired for anonymously allocated memory (i.e., using `MAP_ANONYMOUS` or mapping `/dev/zero`). The only difference is that no file format needs to be designed; it is all up to the program to decide.

2.3.2 File Access Permissions

The user and group ID of the file should be selected to include the smallest number of users. If possible, permissions for group or other users should be disabled completely or at least to only read-access. Programs should not rely on the umask setting of the session. Explicitly disabling access permissions in the `open()` call is the much better alternative.

Instead of widening the permissions if the coarse user and group permissions are not sufficient, it might be desirable to use access control lists (ACLs). ACLs allow to grant rights to arbitrary people on a file by file basis. All that is needed is that it be supported in the operating system kernel and the filesystem. This is all given in Red Hat Enterprise Linux. To take advantage, programs might have to be extended. The `libacl` package provides the necessary APIs to set and remove ACL entries for files or directories. It is necessary to set the permissions when a file is created. With the help of default ACLs which can be defined for directories, explicit work can be avoided if all new files are created in directories which have such a default ACL defined.

ACLs are powerful but also quite complicated. Familiarity with all aspects is needed to take advantage of them correctly and not mess up the security. We will later discuss another way to restrict accesses (SELinux). ACLs have the advantage that they are widely available and also can work with NFS (and AFS?).

The requirement to set the best set of permissions also applies to objects other than files which reside in the filesystem. These other objects include devices, Unix domain sockets, and FIFOs. The `mknod()` system call enables you to specify the access mode as part of the parameter list. If Unix domain sockets are created using `bind()`, a separate `chmod()` call is needed.

2.3.3 Process User and Group ID

The process user and group IDs have a number of different uses, including controlling access to the filesystem, setting permissions to send signals to other processes, setting resources of another process, and other things. In addition the special role of the superuser (ID 0) has to be taken into account.

There are two general pieces of advice:

- use the superuser ID only when absolutely necessary;
- share the user and group ID of a process with as few other processes as possible.

The superuser ID is necessary for a process if the process needs to perform operations which are not available to other users. Operations which need these extra privileges include creation of sockets for port numbers less than 1024, creation of raw sockets, changing file ownership, and many other things. If a process needs to perform any of these operations there is no choice but to execute that code with the superuser privileges at that time. This does not mean the process has to run with the privileges all the time. There are several options to avoid it:

- the root privileges can be installed only when needed. The POSIX functionality of saved IDs helps doing this. One can switch to an unprivileged ID for the normal operation and back to root privileges when needed. The requirement is that the process' initial privileges are those of the superuser.

  This method does not provide full protection since an intruder could insert code into the process which switches the user or group ID to zero and then runs code which uses the superuser privileges.

- restrict the privileges of the superuser. Internally the Linux kernel does not check for user ID zero when a privileged operation is requested. Instead a bitset with bits for various classes of permissions is tested. A process started with superuser privileges has all the bits set. Such a process then can select to remove individual privileges. The header `<linux/capability.h>` contains the necessary definitions. All privileges which are not needed should be disabled.

  The `libcap` package provides the necessary library to modify the capabilities of the process and create arbitrary sets. The `cap_set_proc()` interface installs the constructed bitset.

- the work the program performs can be split in two parts: one process to perform the privileged work, a second one to do the real work (this is called privilege separation). These really need to be two processes, not two threads. To have any effect on security, the privileged process must run different code which has the ability to perform the privileged calls and the communicate with the unprivileged process, but nothing else. This small piece of code can more easily be audited. This will make an exploit harder and in case it still happens, using the exploit is harder since little of the existing code can be used for the exploitation.

  The complicated piece of this framework is the communication between the two processes. It must be secure and not be prevented by the difference in privilege levels. This automatically excludes signals as a communication or notification mean. There are a number of possibilities: pipes, message queues, shared memory with futexes, and more.

  All the other methods mentioned here can be implemented for the privileged process; this way increasing security even more.

  Using separate processes has the additional advantage that it works with SELinux. The state transition required to change the roles which govern SELinux permissions require starting an `exec` call.

  All this, together with the availability of fast interprocess communication mechanisms and Linux's fast context switching, makes using separate processes attractive.

2.3.4 Filesystem User and Group ID

The Linux kernel enables you to select user and group IDs for filesystem accesses that differ from the process user and group IDs. This allows the removal of the capability of accessing the filesystem while keeping superuser privileges for other purposes.

```
int setfsuid(uid_t fsuid);
int setfsgid(uid_t fsgid);
```

A network daemon which needs superuser privileges to create special sockets can create files as an ordinary user and read/write only to files this ordinary user has access to.

2.3.5 File Descriptor Mode

A file descriptor can be restricted in the operations that it can perform. The primary restriction is allowing the use of the descriptor for reading, writing, or both. A descriptor for a file which needs only be read should not be open for writing. This would only allow an intruder to wreak havoc on the file. Note that this recommendation requires the use of the optimal protection mode for `mmap()` calls as well, in case the descriptor is used for mapping.

In general, if a file is written to, the content might also have to be read. So the read/write mode is the most likely mode if read-only is not an option.

In some situations it makes sense to have a file descriptor open only for writing. This is useful, for instance, when creating log files. Using only writing prevents information leaks since, if permissions are set up appropriately, the file can be prevented completely from being read.

In this last situation it is usually a good idea to specify the O_APPEND flag as well. This flag enforces that all write accesses using the file descriptor always append to the file. This is regardless of the current position of the file offset for reading. The result is, obviously, that no content of the file can be overwritten.

When using file descriptors, the programmer should chose the least privileged mode. If a file descriptor is mostly used for reading but occasionally also for writing, it might be worth considering two different descriptors. The descriptor for reading could be kept open all the time, and a new descriptor which also allows writing, would be opened on demand. It is not possible to change the mode of a descriptor. Reopening a descriptor with a different mode is quite easy. The following code can replace an existing descriptor with a new one which allows writing as well:

```
int reopen_rw(int fd) {
  char buf[40];
  sprintf(buf, "/proc/self/fd/%d", fd);
  newfd = open(buf, O_RDWR);
  dup2(newfd, fd);
  close(newfd);
  return fd;
}
```

It is of course also possible to just open an additional descriptor, use it while write-access is required, and then close it. This example shows that you can have a read-only file descriptor open only when it is really necessary. Note that the use of the /proc filesystem ensures that both descriptors really use the same file.

2.3.6 Resource Allocation

An intruder can cause problem by negatively effecting the operation of the other processes using the system. If a process is allowed to exhaust resources shared with other processes, the other processes are starved of those resources. For instance, if the intruder can use up all the RAM in the machine, all other processes would be swapped out and their performance would degrade significantly. This performance loss of an application might even open a window for an attack if the process cannot react quickly enough to prevent an attack.

For this reason, and application should be allowed the resources to work perfectly, but limits should be set to catch "runaway processes" that use too many resources.

The resources which can be easily restricted include memory usage, stack size, number of file descriptors, file locks, POSIX message queues, and pending signals. The resource limits are set using the setrlimit() interface.

To determine a usable limit is not easy. It is possible to track open(), socket(), and other such calls to determine the biggest descriptor used. With some knowledge of the program it is then possible to select a limit. In general, monitoring an application and its resource usage will be necessary. There is so far no tool for just this purpose; maybe this will change in future.

2.4 Do Not Trust Anyone

Whenever possible, check whether the communications a program receives really comes from a program which is an expected communication partner. Secure communication protocols have this authentication build in. This is not covered here (so far). There are some OS services which provide some level of authentication which is independent of the communication protocol.

2.4.1 Authentication via Unix Domain Socket

If two processes (obviously on the same machine) communicate via Unix Domain sockets, it is possible for those processes to authenticate each other by retrieving the process, user, and group ID of the other process. This can be achieved with a call to getsockopt() after the first data has been received from the socket:

```
struct ucred caller;
socklen_t optlen = sizeof(caller);
getsockopt(fd, SOL_SOCKET, SO_PEERCRED,
           &caller, &optlen);
```

If fd is the descriptor for a Unix domain socket, the getsockopt() call will fill in the pid, uid, and gid elements of the ucred structure. This information can then be used for authentication. If the mere numbers are insufficient, it is possible to look at the information in proc/<pid>/ (where <pid> is the process ID received in the getsockopt() call).

The information in the ucred structure is filled in by the kernel. A process is able to set the user and group ID values to whatever would be allowed for a setuid() or setgid() call. This is done with code like this:

```
struct iovec iov[1];
iov[0].iov_base = "foo";
iov[0].iov_len = 4;
char b[CMSG_SPACE(sizeof(struct ucred))];
struct msghdr msg =
{ .msg_iov = iov, .msg_iovlen = 1,
  .msg_control = b,
  .msg_controllen = sizeof(b) };
struct cmsghdr *cmsg =CMSG_FIRSTHDR(&msg);
cmsg->cmsg_level = SOL_SOCKET;
cmsg->cmsg_type = SCM_CREDENTIALS;
cmsg->cmsg_len
  = CMSG_LEN(sizeof (struct ucred));
```

```
struct ucred *uc
  = (struct ucred *) CMSG_DATA(cmsg);
uc->pid = getpid();
uc->uid = someuid;
uc->gid = somegid;
msg.msg_controllen = cmsg->cmsg_len;
sendmsg(fd, &msg, 0);
```

Usually this should not be necessary. The default values returned by the kernel in the absence of a previous `sendmsg()` call (like that above) are accurate for the two participating processes.

2.4.2 Signal Origins

A process can receive a signal from another process and act accordingly. If the signal handler executes the code unconditionally this means the code can be initiated from another process. In many cases the signal handler is only meant to catch signals generated in the same process. For instance, a signal handler to catch segmentation faults should never be invoked by a signal sent from the outside.

The kernel performs checks before allowing a signal to be sent. Only root and a process with the same current or effective user ID as the target process is allowed to send a signal. If such a process has been compromised, this is of no help.

The traditional Unix signal handler model does not have support for any checks. It is necessary to register signal handlers which used the new `siginfo` interface. The type for such a signal handler is this:

```
void (*siginfo_handler) (int, siginfo_t *,
                         void *);
```

The interesting part here is the second parameter. This structure contains a number of fields, chief among them `si_signo` and `si_code`. The `si_signo` contains the signal number, the same value as the first parameter to the signal handler. The `<signal.h>` header defines a number of values the `si_code` can have. There are a number of values which can be used with every signal. If the `si_code` does not have any of these values, the value must be interpreted relative to the signal number. The POSIX specification defines which values are defined for which signal.

The independent values are the interesting ones here. They have the prefix `SI_` and help to distinguish explicitly generated signals from those created implicitly by, for instance, a segmentation fault. The explicit generation codes are:

`SI_USER` Sent by `kill()`, `raise()`.

`SI_QUEUE` Sent by `sigqueue()`.

`SI_TIMER` Sent by timer expiration.

`SI_MESGQ` Sent by message queues.

`SI_ASYNCIO` Sent by AIO completion.

`SI_SIGIO` Sent by `SIGIO`.

`SI_TKILL` Sent by `tkill()`.

A segmentation fault handler which checks whether the signal was really raised by the kernel in response to an invalid memory access could start like this:

```
void handler(int sig, siginfo_t *infop,
             void *ctx) {
  if (infop->si_code <= 0)
    return;
  ...
}
```

This hardcodes the condition that all `SI_*` values have a value less or equal to zero. This is true for Linux. Alternatively this could be used:

```
void handler(int sig, siginfo_t *infop,
             void *ctx) {
  if (infop->si_code != SEGV_MAPERR
      && infop->si_code != SEGV_ACCERR)
    return;
  ...
}
```

This hardcodes the (equally true) assumption that these are to only two codes defined for `SIGSEGV` signals. If the same signal handler would also be used for other signals, a test for the value of `sig` or `infop->si_signo` would be needed, too.

2.4.3 Avoid Sharing

Some directories in the filesystem are shared, such as `/tmp`, and programs (and the C library) use this directory quite often. The problem is that everybody could create a file which another user's programs might use. This is a problem in the program which reads these files created by others, but it is a problem which can be avoided.

One possibility is to avoid using the actual `/tmp` directory, for instance. The Linux kernel allows processes and their children to have separate filesystems mounted. Processes can have their own filesystem namespace, together with the "bind mounts", the ability to mount parts of a filesystem in another part of the filesystem. To provide a separate `/tmp` for each user, the following code could be used:

```
int f (void *arg) {
  const char *home = getent("HOME");
  char buf[strlen(home)
           + sizeof "/my-tmp"];
  sprintf(buf, "%s/my-tmp", home);
  mount(buf, "/tmp", 0, MS_BIND, 0);
  int fd = creat("/tmp/foo", 0600);
  ...
  exit(0);
}

int main(void) {
  char st[50000];
  pid_t p = clone(f, st + sizeof(st),
                  CLONE_NEWNS, 0);
  exit( p == -1 ? 1 : 0);
}
```

This code would not appear in this form in any program; it just shows the concept. The `clone()` call is basically a `fork()` but the `CLONE_NEWNS` flag introduces a separate filesystem namespace for the child process. The child process is executing function `f()` which then mounts the directory `my-tmp` in the user's home directory as `/tmp`. From then on, any filesystem access to a file such as `/tmp/foo` actually accesses `my-tmp/foo` in the home directory. All programs work as before but the two users' data in `/tmp` does not conflict. Since modifiable data which is supposed to be shared between multiple processes and users should be stored somewhere in the `/var` hierarchy, using temporary directories like this is really a good possibility. Another possibility is that an application might chose to create a temporary directory for each instance and then mount this directory in a certain fixed place in the filesystem tree.

2.5 Truly Random

Programs need some randomness for various purposes. Unique file names and identification numbers are needed all the time. Communication protocols might require sequence numbers which, to reduce the possibility of attacks, should be random. Programs are sometimes inherently random (simulation, games) and need large amounts of randomness.

For all these purposes people over time found numerous thoroughly stupid ways to get "randomness". For instance, temporary file names were constructed using the process ID. The fact that the process ID is not random was of no concern. People tried to read random parts of the address space and use the bytes found there. All this is insufficient and even a security risk if the numbers are used in any security algorithm.

The standard runtime has had support for random number generation from the beginning. The problem is that there are several different interfaces and the interfaces can, in most cases, be implemented differently on different platforms. And there is always the problem of seeding the pseudo random number generator (PRNG).

Some of the implementations of those PRNGs are horrible. The POSIX specification, for instance, requires an implementation of `rand_r()`:

```
int rand_r(unsigned int *seedp);
```

The whole state this PRNG can have is in the `unsigned int` value the parameter points to. These are 32 bits on Linux platforms which is a ridiculously small amount. This function should *never* be used. A better interface is the `random()` family:

```
long int random(void);
void srandom(unsigned int seed);
char *initstate(unsigned int seed,
                char *state, size_t n);
```

On Linux, the `random()` function returns a 31-bit random number. The period of the default PRNG used is very high, it is a trinomial, not a simple linear one. The state used can be changed with the `initstate()` interface. The state of the PRNG can be increased up to 256 bytes. There are also reentrant versions of these interfaces available. A critical part of the use of the interface is to use a good seed, either for `srandom()` or `initstate()`. More about this later.

The `rand48` family of functions is another usable possibility. As the name suggests, the state of these functions has 48 bits. There are two main benefits of these interfaces: the implementation is the same on all Unix systems (i.e., with the same seeds one gets the same results on different platforms). The second benefit is that one cannot only get 31-bit integer random numbers. It is also possible to get floating-point numbers and 32-bit signed integer values. The drawbacks include the limited state size and the fact that the algorithm is required to be a simple linear one.

If PRNGs must be used, `random()` should be preferred. It is possible to construct floating-point values from the provided integer values. If this is too cumbersome or reproducibility on other platforms is required, the `rand48` functions are better.

If only small amounts of randomness is needed, modern systems provide another possibility. The kernel provides two special character devices `/dev/random` and `/dev/urandom`. These are character devices which can be read. There is a qualitative difference between the two devices: `/dev/random` provides higher quality data, as close to true random data as one can get without special

devices. The disadvantage is that reading numbers from this device might block if there are not enough random numbers left. The block can block for a long time if there are no events which can be used to generate randomness.

Calls to read from `/dev/urandom` on the other hand will never block. Cryptographic algorithms are used to generate data if the system runs out of true random data. The numbers provided might therefor classify as pseudo-random but are usually of a higher quality than the purely algorithmic methods available in the standard runtime.

Retrieving large amount of randomness from these devices is not advised. If many calls have to be made, retrieving the values is slow since each time a system call is needed. More importantly, randomness in the kernel is a shared resource. If the pool is exhausted, some `read()` calls might block. This could in fact lead to denial of service attacks.

There is no problem with reading a few bytes of high-quality random bytes. If this is enough for the program, fine. Otherwise it might be desirable to use these bytes to seed the PRNGs in the standard runtime.

There is one more possibility which is becoming more and more popular. The system itself might have devices to generate random data. Some Intel chipsets can create such data but it cannot be used directly. Instead, a special device driver feeds the data into the pool for `/dev/random`. Recent VIA processors (IA-32 compatible) have a built-in true RNG. With just a simple instruction some random data can be retrieved. If it is OK to write code for special processors, this is the way to go for these processors.

2.6 Automatic Fortification

Starting with Red Hat Enterprise Linux 4, programs can be compiled with the macro `_FORTIFY_SOURCE` defined. As with all other `_*_SOURCE` macros, it should be used on the command line of the compiler. When this macro is defined, the C compilation environment is transparently changed a bit. Various security checks are enabled which have one thing in common: they have no or no significant cost at runtime. The reasoning behind this is that these tests can be enabled for all programs all the time.

The macro `_FORTIFY_SOURCE` must be defined as a numeric value. Currently two setting are supported, 1 and 2. In mode 2 all tests of mode 1 are performed, and then some more. The definition of the macro has limited effect if optimization is not enabled. The reason is that some of the information propagation inside the compiler does not happen unless optimization is enabled.

The tests which are currently performed fall into two categories:

- functions operating on memory blocks are checked if the size of the memory blocks is known.
- ignoring return values of functions for which this should not happen is flagged.

Further tests will likely be developed in future and the two existing test sets will be extended. For instance, not all headers of the standard C runtime have so far been annotated to provide warnings about unused return values.

The return value tests are pretty simple: the GNU compiler allows adding an attribute to the function declaration for just this purpose. It helps to identify incorrect code like this:

```
if (req > len)
  realloc(p, req);
memcpy(p, mem, len);
```

The code assumes that the enlarged buffer `realloc()` makes available is at the same location the old buffer was. This might even be true in many situation and the programmer could run all the tests without noticing. In general this is not the case and therefore the above code will fail sooner or later. Other examples of problems detected are possible security problems.

The memory checking tests are more elaborate. In section 4.1 we will discuss various tools to debug memory handling more thoroughly. All these tools have one thing in common: they are expensive to use or even do not integrate into the final program at all. The memory debugging code enabled by `_FORTIFY_SOURCE` is cheap at runtime.

Currently the memory checks are restricted to functions of the standard C runtime. So far these functions are handled:

`bcopy()`	`fgets()`	`fgets_unlocked()`
`fprintf()`	`getcwd()`	`gets()`
`getwd()`	`memcpy()`	`memmove()`
`mempcpy()`	`memset()`	`pread()`
`pread64()`	`printf()`	`read()`
`readlink()`	`recv()`	`recvfrom()`
`sprintf()`	`stpcpy()`	`strcat()`
`strcpy()`	`strncat()`	`strncpy()`
`vfprintf()`	`vprintf()`	`vsprintf()`

Not all calls to these functions are checked. This is not possible since in general there is no information about the buffers available. There are some situations where the compiler knows about the sizes:

- when the buffer is an automatic one, i.e., allocated on the stack. Such a buffer is created by the compiler. This includes calls to `alloca()`.
- when the buffer was allocated in the function. The compiler knows that functions like `malloc()` are

used for memory allocation and it knows the parameter which specifies the size.

In gcc 3.3 this information is not used. The compiler lacks the infrastructure. gcc 4 has support to propagate this information.

If the conditions of the known buffer size is met, it might be possible that it can deduce that there will always be a buffer overflow. For example, compiling this code

```
#include <string.h>
void foo(void) {
  char buf[4];
  strcpy(buf, "hello");
}
```

will produce the following warning:

```
In function 'foo':
warning: call to __builtin___strcpy_chk
 will always overflow destination buffer
```

In any case, the compiler will replace the call to `strcpy()` with a call to `__builtin___strcpy_chk()` if the target buffer size is known. Similarly, the calls to the other functions are replaced. The `__builtin_*` functions are part of the GNU C library. A programmer need not know about them. All this happens transparently and almost without side effects. The functions are only immeasurably slower than non-checking normal functions. The effects of the checking can be seen by compiling this code:

```
#include <string.h>
int main(int c, char *v[]) {
  char buf[5];
  strcpy(buf, v[1]);
  return 0;
}
```

In this code the compiler cannot detect any overflow at compile time. But it knows the size of the target buffer and there is the potential for a buffer overflow, so it generates a call to the checking function. If this program is started with parameter which is longer than 4 characters the following message is printed and the program terminates:

```
*** buffer overflow detected ***: prg terminated
```

Terminating the program is really the best option because the program becomes unstable when memory is modified in a way the program does not expect. Buffer overflows are one of the main means for attackers to take control of a system.

The difference between mode 1 and 2 of the fortification code is that mode 2 makes more strict assumptions. For instance, the return value checks are only enabled in mode 2. Another difference is the handling of code like this:

```
#include <string.h>
struct {
  char a[10];
  char b[10];
} f;
void foo(void) {
  strcpy(f.a, "012345678901");
}
```

When compiled with fortification level 1 the compiler does not produce any warning since this is silly, but, strictly speaking, legal code according to ISO C. With level 2 the usual overflow warning is printed. It is best to avoid such code and therefore the recommendation is to compile all code with `-D_FORTIFY_SOURCE=2`.

3 Good Practices in Compilation

The GNU C compiler has more capabilities than most other compilers. Among them is that it integrates the capability to analyze the code while compiling. Other compilers do not have this ability and need a separate tool called lint. The GNU compilers and linker also allow you to annotate function declarations and definitions so that their usage can be monitored.

In this section we will show several ways in which the compiler and linker can help you write good code.

3.1 Respect Warnings

Probably the most important advice for using the compiler is to enable all generally accepted warnings and eliminate them. The warnings concerning problems with the source code are enabled by adding the compiler options

```
-Wall -Wextra
```

These options are available for all languages which the GNU compilers can handle. There are additional warnings which could be enabled but these are either useful to determine inefficiencies (mostly to determine unused

variables and parameters) or controversial. The latter would generate too many false positives if enabled without close inspection of the source code.

For C++ code one should use

```
-Wall -Wextra -Weffc++
```

The additional warnings this new option enables are C++-specific and are derived from the buidelines in the books "Effective C++" and "More Effective C++", hence the name. It is highly recommended to use `-Weffc++` since the problems the compiler warns about are in most cases bugs.

When compiling many files using make or similar tools, it is easy to miss a warning. Therefore it is useful to have the compilation fail when a warning is printed. This is when the option

```
-Werror
```

comes in. It causes the compilation to fail if a warning is emitted. If existing code has to be converted to compile with these three options, a lot of work might be necessary at first since even good programmers have one problem or another discovered by the compiler in their code. It is best to start using the flags right from the start to not accumulate problems.

There is a problem, though. Some code cannot easily be changed to compile without warnings. This is especially true for generated sources. For instance, code generated by flex and bison often does not compile cleanly. The Makefiles used in the project therefore should allow disabling the `-Werror` option for individual files.

3.2 Avoiding Interfaces

The standard C library contains a number of functions which are unfortunately required by standards but should not be used. We have seen two examples earlier: `gets()` and `getwd()`. The compiler and linker will warn about using these interfaces when the GNU C library and the GNU linker are used. The tools do not recognize these function names. The technique to mark these functions can be used for every interface. Library authors should use these techniques to mark all deprecated or dangerous interfaces.

In header files the `deprecated` attribute should be used. As in:

```
extern int foo(int)
  __attribute((__deprecated__));
```

This attribute will not do anything if the declaration is just parsed. Only when the function is used will a warning be emitted.

In addition a linker warning is useful. If the object file is distributed and used at a different time, the compilation warning is obviously never seen. Therefore the object file must contain the warning information. The extra information has to be generated at the time the object file is generated and added to the file. Continuing the example above, one would add the following after the *definition* of the function:

```
__asm(".section .gnu.warning.foo\n\t"
      ".previous");
static const char foo_deprecated[]
  __attribute((unused,
    section(".gnu.warning.foo")))
  = "Don't use foo";
```

This might be a bit hard to understand if one is not accustomed to ELF, sections, and the syntax used. The `asm` statement (a GNU compiler extension) simply introduced the section named `.gnu.warning.foo`. The following definition of the variable `foo_deprecated` places the variable in this special section. This is achieved using the attribute. The section is not defined to be allocated in the output file. That means, even if the functions and variables in the file are included in the resulting executable, the warning string is not. The warning is included when creating a DSO, since otherwise one cannot get a warning when linking with the DSO.

We have not explained how the warning is actually emitted. The magic is the section name. For all sections starting with `.gnu.warning.` followed by the name of a function, the linker prints the content of the entire section as a warning. If there would be a reference to `foo()`, the linker would find the section `.gnu.warning.foo` and the content of this section is the string `"Don't use foo"`.

3.3 Unused Return Values

All C interfaces and many C++ interfaces provide error status in the return value of the function. This makes it all too comfortable to not check the return value. Adding many `if`s takes effort and if the programmer thinks the function will never fail, this might lead to deliberate omissions of the check. The result will be a lot of problems of many kinds.

Security problems can arise if, for instance, the return value of `setuid()` is ignored. Semantic problems are ignoring the return value of `realloc()`; somebody might assume that the buffer is enlarged in place, and that the new buffer has the same location as the old one.

Since it is unreasonable to expect all return values to be

checked, a library author should mark those interfaces, for which the omission is critical. The declaration should be marked as follows:

```
extern int foo(int arg)
  __attribute((__warn_unused_result__));
```

If a call to `foo()` is performed after seeing this declaration and the return value is not used, the compiler will issue a warning. Note that casting the result to `void` does not help; the warning is still issued.

3.4 Null Pointer Parameters

Manys function, which have parameters of a pointer type, expect the pointer values passed in a call to not be `NULL`. Examples are the string functions like `strcpy()` which will mercilessly crash the process if any of the parameters is not a valid pointer. The GNU C compiler provides a way to detect `NULL` values to the function. The annotated prototype for `strcpy()` looks like this:

```
extern char *strcpy(char *dst,
                    const char *src)
  __attribute((__nonnull__(1, 2)));
```

The `nonnull` attribute has a parameter list with numbers representing parameter position to which the attribute applies. In this case parameter 1 and 2 are required to be non-`NULL`.

The checking applies only to cases the compiler can decide at compile time. The warning is only printed if the expression passed as the pointer parameter is evaluated to zero. If the value passed is variable, no warning is printed and nothing special happens at runtime if a `NULL` pointer value is actually passed. The called function will use the `NULL` pointer and fail.

Marking functions with pointer parameters with this attribute will not necessarily catch many bugs since the values passed in the parameters of the annotated functions are more likely be variables. But it is nevertheless a useful annotation because experience shows it can detect problems. Sometime parameters are transposed, function names are misspelled or similar names are used.

4 Debugging Techniques

Unfortunately debugging is a large part of a programmer's life. Fortunately several tools have been developed to support debugging. The following section describes some of the available tools.

4.1 Memory Handling Debugging

Memory handling bugs are still very common. These bugs are also hard to be reproduce and find. Several tools are available for various debugging stages.

4.1.1 Runtime Tests

The `malloc()` implementation in the GNU C library has always been very sensitive when it comes to memory handling bugs. Buffer overruns often crash applications. In the most recent versions, a number of additional tests have been added so that bugs are discovered earlier. Programs which have run nicely before now might stop working. This is due to bugs in the memory handling. The messages GNU C library emits are of this form:

```
*** glibc detected ***: prg: double free or
  corruption: 0x000000000079d4e0
```

followed by information about the program state (if available). There are a number of different reasons, the text after the second `***` explains them. Finding the actual reason for the problem is another matter. There are basically three problems:

- invalid pointer passed to `free()`. This can mean pointers to an object on the stack, in static memory (data sections), or in memory allocated with `mmap()`. It can also mean a pointer in the middle of a block allocated with `malloc()`.
- `free()` called more than once for a pointer. This might also happen implicitly, e.g., when calling `fclose()` twice on the same stream.
- buffer overruns.

The message printed should give a hint as to what the `malloc()` implementation thinks the problem is. Invalid pointers should be easy to detect. The `mtrace` handling described below helps to trace allocations and then detect invalid pointers. Similarly double `free()` calls are detected this way.

The reasons for buffer overruns are much more difficult to detect. The problems are usually detected after the buffer overrun occurred, sometimes much later. If tools like `mudflap` or `valgrind` cannot detect the problem, one needs to do a lot of research. One possibility is to compile code with the definitions from appendix A. This will allow tracing back a memory place to the place where it was allocated. This might not show where the buffer overrun happens, but the allocation location and the bytes written during the overrun often provide enough clues.

For instance, a bug in `rpm` was detected by the `malloc()` implementation. The buffer meta data, which was used in the detection, had the value 66636539_{16}. This looks like ASCII data. Investigating the memory before this word shows this:

```
DSA/SHA1, 2004.....Key ID da84cbd430c9ecf8
```

This quickly lead to one specific place in the source where the `DSA/SHA1` was used. In this case, as in many others, the bytes which have been written beyond the buffer boundaries often helps locating instruction causing the buffer overrun. The artificial buffer annotations can help if the identification of the source is not that easy. Often it is sufficient to look at the annotation of the previous block in memory since modifying that block is usually what caused the buffer overrun to happen.

4.1.2 Freeing Everything

The runtime tests and memory leak tests work their best only if the program is freeing all the memory it allocated. The `free()` function has quite a few of runtime tests which recognize problems. Distinguishing memory leaks from regular allocations which are used for the duration of the program run is hard unless all memory is freed. Declaring all non-freed memory an error makes detecting memory leaks easier.

Always freeing all memory blocks is–for normal, non-debugging program runs–a waste of time. It unnecessarily slows down the termination of the process and therefore affects system performance. On the other hand it is highly undesirable to have separate binaries, or even sources, for debugging.

A solution is to include the code to free the memory only when explicitly requested. This can be triggered by an program option, environment variable, or "magically" initiated by a memory testing tool (`valgrind` does this for the memory allocated in the GNU C library).

It is a big hassle, though, to write all these functions to free the memory in the various pieces of the program and then call them without missing a function.

One solution for this problem is to determine the set of functions which need to be called automatically. There are several methods; one which has been proven useful is to use ELF sections. When defining a cleanup function we also define a pointer to this functions and store it in a special section. Then, when we have to call all cleanup functions we iterate over all the function pointers in this special section and call the functions they point to one after the other. A possible solution could then use the following macro and definition:

```
asm (".section cleanup_fns, \"a\"");
#define cleanup_fn(name) \
  static void name(void); \
  static void (*const fn_##name)(void) \
    __attribute((unused,
     section("cleanup_fns"))) = &name; \
  static void name(void)
```

The macro can then be used like this:

```
static const char *myname;

cleanup_fn(free_myname)
{
  free(myname);
}

void set_myname(const char *s) {
  free(myname);
  myname = strdup(s);
}
const char *get_myname(void) {
  return myname;
}
```

The function `free_myname()` is statically defined and a pointer to it is stored in the section `.cleanup_fns`. The name of the variable used to store the pointer is unimportant; it will never be used. Note that `myname` is defined with file-local scope; it need not be visible outside the file since the cleanup function is defined in the same file. The name of the section it is defined in has no influence.

What remains is to specify how the cleanup functions are called. For this we use a nice little feature of the GNU linker. The GNU linker automatically generates two symbols for a section if

a) the section name is a valid C identifier

b) the symbol is referenced

Looking back at the macro definition we see that the name of the section for the function pointers is indeed a valid C identifier. Now all we have to do is to reference the magic symbols.

```
extern void (*const __start_cleanup_fns)
  (void);
extern void (*const __stop_cleanup_fns)
  (void);
```

```
void run_cleanup(void)
{
  void (*const *f)(void)
    = &__start_cleanup_fns;
  while (f < &__stop_cleanup_fns)
    (*f++)();
}
```

The names of the two "magic" symbols are constructed by the linker by prepending `__start_` and `__stop_` respectively to the section name. The two declarations at the top of the code above declare these two variable. The two variables define the start and end end of the section. Since the section contains function pointers, all we have to do is to iterate over the content of the section and call the functions. The syntax in `run_cleanup()` might be a bit confusing, but it is really a trivial function.

There is one more thing which should be mentioned in this context. It often happens that many of the cleanup functions look like the `free_myname()` function above which consists of only a call to `free()`. If this is the case it is worthwhile making this a special case: instead of requiring a function to be defined we simply put the pointer variable, which needs to be freed, in a special section and iterate over this section.

```
asm (".section cleanup_ptrs, \"aw\"");
#define cleanup_ptr(name)  name \
 __attribute((section("cleanup_ptrs")))
```

The example to use the cleanup mechanism we saw above can now be rewritten as follows:

```
static const char *cleanup_ptr(myname);

void set_myname(const char *s) {
  free(myname);
  myname = strdup(s);
}
const char *get_myname(void) {
  return myname;
}
```

Note that even though we define `myname` as a special case, it is used just like any other variable. It only is special when it comes to cleanup. The complete definition for the cleanup handler now has to look like this:

```
extern void (*const __start_cleanup_fns)
  (void);
extern void (*const __stop_cleanup_fns)
  (void);
extern void *const __start_cleanup_ptrs;
extern void *const __stop_cleanup_ptrs;

void run_cleanup(void)
{
  void (*const *f)(void)
    = &__start_cleanup_fns;
  while (f < &__stop_cleanup_fns)
    (*f++)();
  void *const *p = &__start_cleanup_ptrs;
  while (p < &__stop_cleanup_ptrs)
    free(*p++);
}
```

The function is still trivial, there are now just two loops. The second one calls `free()` for all the pointers in the section. This will work even if the variable has not been used in the program run since `free(NULL)` is a no-op.

With these kind of definitions it is trivial to free all memory. The programmer just has to remember to use the `cleanup_ptr` or `cleanup_fn` macro and conditionally call the `run_cleanup()` function. There are no penalties to be paid in terms of performance when the memory need not be freed.

4.1.3 Uninitialized Memory

Another source of problems with memory handling is uninitialized memory. Variables allocated by the compiler in the data sections are always initialized, but variables allocated on the stack or heap are not. Reading those variables and using the value before the first write operation leads to unpredictable results. Some people like to explicitly initialize all memory after allocation but this is a big waste of time when the program gets executed. It is possible to avoid this waste and still avoid using uninitialized memory.

In case memory is allocated dynamically and should have an initial value of all zeros, one should use `calloc()` instead of `malloc()`. Not only does this make sure that all memory is initialized, not more and not less, the implementation can also avoid the actual initialization process if it can determine the memory is already cleared.

Detecting whether all memory is initialized is not easy. The `valgrind` tool (see page 23) can help but it is slow. The C library can help determining problems with dynamically allocated memory. If at runtime the environment variable `MALLOC_PERTURB_` has a numeric value, or the function `mallopt()` is used to set the `M_PERTURB` option, all memory allocated with a function other than `calloc()` is initialized with the given byte value. After a call to `free()` the freed memory is overwritten with another, derived byte value to make sure uses of the now invalid memory can be detected.

There is no guarantee that using these options helps detecting problems. Programs might graciously handle such invalid input. More often than not, the result is that the program behaves in strange ways or simply crashes. The latter is often the case if the memory values are used as pointers. It is therefore useful to use these options during the QA phase since no correct program must be disturbed in its operation by it.

These two options do not help at all with detecting problems with variables allocated on the stack. For detecting these kind of problems the warnings the compiler emits when it finds unallocated code, are the best source

4.1.4 Memory Allocation Hooks

The GNU C library contains two interfaces to help memory debugging. These interfaces use hooks available in the `malloc()` implementation which can intercept the memory handling function calls. This mechanism so far does not work in multi-threaded application. If a single thread is used, these functions are very useful.

The first is `mtrace()`. This function should be called as the first thing in the `main()` function. The function by default does nothing. It requires the environment variable `MALLOC_TRACE` to be set at runtime. The value must name a file into which the output is written. After the `mtrace()` call each memory allocation and deallocation is logged into the output file. This output is not supposed to be read by humans. The files can grow very large, depending on the number of memory operation performed. Once the program terminates (or the `muntrace()` function is called), the program can be processed using the `mtrace` script.

As an example assume the following code:

```
int main(void) {
  mtrace();
  void *p = malloc(10);
  malloc(20);
  free(p);
  return 0;
}
```

When started with `MALLOC_TRACE=mout` in the environment, after the program terminates the file `mout` will contain the following:

```
= Start
@ ./m:[0x80483fb] + 0x811a378 0xa
@ ./m:[0x804840b] + 0x811a388 0x14
@ ./m:[0x8048419] - 0x811a378
```

This data can then be processed by executing the following command: `mtrace program mout`. Doing this for the example results in this:

```
Memory not freed:
-----------------
   Address     Size     Caller
0x0811a388     0x14  at 0x804840b
```

As can be seen, the memory leak in the code was detected. This has been tested to work on big programs with huge numbers of memory allocations with output files of more than 2GB in size. Using the `muntrace()` function is usually not needed since the entire program run should be checked. If only a specific part of the program should be checked, surround the code with an `mtrace()`/`muntrace()` pair.

`mtrace()` has one problem, though: it cannot be used reliably in multi-threaded applications.

The second interface for memory debugging is `mcheck()`. There are actually three separate interfaces:

```
int mcheck
  (void (*afct)(enum mcheck_status));
int mcheck_pedantic
  (void (*afct)(enum mcheck_status));
void mcheck_check_all(void);
```

The first two functions enable the checking mode. After the call, all calls to the `malloc()` functions will perform sanity checks of the blocks involved. In case the `mcheck_pedantic()` function is used, all the blocks allocated at that time are checked. As this can easily be a very slow operation, it should be used with care. If a memory handling error is detected, the function registered with the `mcheck()` or `mcheck_pedantic()` call is called with an appropriate error value.

If these hooks still do not provide a fine enough granularity to determine when the memory allocation occurs, the programmer can insert calls to `mcheck_check_all()` at any time. This will cause all allocated blocks to be checked. The result of all this is that the time of a memory corruption can be pinned down with some precision.

4.1.5 Memory Handling Debug Products

Red Hat Enterprise Linux comes with a number of packages which help debugging memory problems. The range of functionality is wide.

`dmalloc` `dmalloc` is a library which contains alternative implementations of the standard `malloc()` functions. If a program is linked with this library, additional tests are performed at runtime and memory leaks can be

detected. The library code will not detect problems when they happen. Just as with the code we have seen in the previous section, it detects problems after the fact. The code is reliable and should not disturb the program at all.

Every source file should be include `<dmalloc.h>` as the last header. The resulting binary, executable or DSO, then has to be linked with `libdmalloc` or, in case of an application which uses threads, with `libdmallocth`.

At runtime the `DMALLOC_OPTIONS` environment variable needs to be set. If the environment variable is not present, the program is run without any additional checks. To determine the value the environment variable needs to be set to, the `dmalloc` program can be used. For instance, to write the debug output into a file `LOG`, perform internal checks moderately often (every 100^{th} call), and to perform all possible checks, use the command

```
dmalloc -l LOG -i 100 high
```

The output is a shell script fragment which can be directly used or one can simply use the assignment in the program start.

```
DMALLOC_OPTIONS=debug=0x4f4ed03,inter=100,\
  log=LOG ./prg param1 param2
```

The output in the log file shows all kinds of information from the amount of memory used, the number of calls made, to the list of memory leaks. In case a memory corruption is detected, the program stops right away with a message.

ElectricFence A completely different approach is taken by `ElecticFence`. It tries to detect memory access problems as they happen. To do this, it always uses `mmap()` to allocate memory. Since `mmap()` can only allocate memory with page granularity, a lot of space is wasted. By default, the actual pointer returned is

$$ptr + \text{roundup}(size, pagesize) - \text{roundup}(size, alignment)$$

I.e., the return value is chosen so that the first byte[2] beyond the requested buffer is on a different page. If the environment variable `EF_PROTECT_BELOW` is defined in the application's environment, the returned pointer is to the beginning of the mapped area, thus protecting against memory accesses with negative offsets.

[2]Due to alignment constrains, there can actually be a few bytes above the object which are not on the next page.

Now all the library has to do is to ensure that the memory on the next or previous page can never be accessed. This can be done by changing the access permissions with `mprotect()` to `PROT_NONE`. The address space following the page with the data therefore has to be sacrificed as well. On 32-bit platforms this can quickly lead to exhaustion of the available address space. The result is that every access beyond the top end of the array will cause the program to crash.

It is also possible to recognize uses of already freed memory. If the `EF_PROTECT_FREE` environment variable is set to 1, address space is never returned to the system. A `free()` call simply makes the memory inaccessible so that future, invalid use of the memory causes a segmentation fault. The drawback is that a program might very fast run out of address space.

There are several limitation a user of this library has to be aware of:

- access checks can only be performed in one direction: either below or above the allocated buffer. The default is to check access violations above the buffer since wrong accesses in the other direction are much rarer but they still something which happens in the real world. A bit better protection can be achieved by running the problem twice, once with and once without `EF_PROTECT_BELOW` defined.
- if the address is only slightly wrong and the block is aligned at the top of the page, some accesses beyond the top of the object might go unnoticed if object size and alignment require a few fill bytes.
- if the address is wrong by an amount greater than one page size, the access might be beyond the guard pages with `PROT_NONE` access created by the library and consequently it might succeed.
- since at least one page is allocated for every memory allocation request, however small it is, the system memory is exhausted much earlier than with the real `malloc()` implementation. In addition, on 32-bit systems the address space size might play a factor. An allocation of 20 bytes would require 12kB of address space (and 4kB of physical memory) on an IA-32 machine.

To use `ElecticFence` there are two possibilities. A program can be linked against `libefence`. Such an executable should never be deployed since the memory allocation is just too inefficient for production use. Alternatively the `ef` script can be used. Simply prepending `ef` to command line of the application. All uses of the `malloc()` functions are transparently replaced with the `ElecticFence` functions by using `LD_PRELOAD`. The program runs normally and does not have specially prepared for using `ElecticFence` in this mode. This makes `ef` easy to use for testing when one suspects an existing program has memory handling problems.

If `ElectricFence` works, it is good and relatively fast. But if the limitations are hit, it becomes unusable. There is no support to find memory leaks, this has to be dealt with in other ways.

valgrind The most useful memory debugger of those presented here is `valgrind`. It is of no use to try duplicating the documentation here. The package comes with the Red Hat Enterprise Linux and Fedora Core distribution and should be installed as part of the developer option. The package contains some documentation which should be read by interested developers.

In short, `valgrind` allows you to debug an unmodified program. It does that by executing the program not using the processor hardware, but instead in a virtual machine. `valgrind` knows about the memory allocation functions in the C runtime and keeps at any one time a complete picture of the memory blocks which are allocated and information whether the data has been initialized or not. While executing the program, each instruction which accesses memory is checked and errors are detected. Not only does `valgrind` detect invalid write accesses (e.g., buffer overflows), it also can detect reads of uninitialized memory. This last feature is particularly useful since not many other tools provide this functionality.

`valgrind` has some limitations, though:

- not all invalid memory accesses are detected, specifically, `valgrind` does not know about object sizes. For instance, assume two arrays which are allocated in memory next to each other. If a write access using a too-big index for the first array will touch the memory of the second array, `valgrind` cannot detect this. The memory written to is allocated.
- to support debugging multi-threaded application, `valgrind` has its own thread library implementation. This is a pure userlevel implementation which is not meant to be fast. The problem is that not only is it not fast, it also does not match 100% the POSIX specification. Programs which might work nicely when using the system's `libpthread` can have problems when run under `valgrind`.
- `valgrind` internally uses heuristics for most tests. While this usually works out, there are false positive reports (and maybe also missing negative reports). One must keep this always in mind: a clean `valgrind` run does not mean the program has no memory handling problems.
- so far `valgrind` is available in production quality only for IA-32.
- emulation is slow, many times slower than native execution (currently 25 to 50 times). This can make it impractical to debug large programs this way. The memory usage also increases by factors due to all the information which needs be maintained to decide whether a memory access is valid or not.

To use `valgrind` one simply has to prepend the program name and the parameter list with `valgrind` and an option to select the wanted tool. Today `valgrind` supports eight different tools. The most thorough memory testing is performed with the `memcheck` tool. The complete command line would therefore look like this:

```
valgrind --tool=memcheck ./prg param1 param2
```

By default only memory access errors are reported and a summary about message use is printed. The program is not terminated when an error is detected. The operation is execution without consideration of the validity and the program continues to run. If `--leak-check=yes` is also passed to `valgrind`, it will also print information about memory leaks. The `memcheck` is pretty slow since it has to check every memory access. The `addrcheck` tool is not as thorough, but is faster.

mudflap Yet another different approach is taken by `mudflap`, a project which is part of the GNU compiler and is ported to all architectures. It is available in the `gcc4` preview packages available in Red Hat Enterprise Linux and Fedora Core. Programs have to be compiled with a new option which adds instrumentation to the code to detect memory handling problems. Assume the following code:

```
int a[10];
int b[10];

int main(void) {
  return a[11];
}
```

When this code is compiled and run using `valgrind`, no errors are detected. The GNU C compiler allocates memory for `a` and `b` consecutively and the access to `a[11]` most likely will read the value of `b[1]`. This access will not fail but the code is clearly not correct. If this code is compiled with

```
gcc4 -o bug bug.c -g -fmudflap -lmudflap
```

and then executed, the program will fail by default and the `mudflap` runtime prints warnings like this:

```
*******
mudflap violation 1 (check/read):
  time=1102895136.630933 ptr=0x600e40 size=48
pc=0x2a95674c88 location='bug.c:5 (main)'
      /usr/lib64/libmudflap.so.0(__mf_check+0x18)
         [0x2a95674c88]
      ./bug(main+0x7a) [0x400992]
      /usr/lib64/libmudflap.so.0(__wrap_main+0x52)
         [0x2a956756c2]
Nearby object 1: checked region begins 0B into and
ends 8B after mudflap object 0x601370: name='bug.c:1 a'
bounds=[0x600e40,0x600e67] size=40 area=static
        check=3r/0w liveness=3
alloc time=1102895136.630865 pc=0x2a95675618
number of nearby objects: 1
```

What this says is that the code in line 5 in `bug.c` performed an incorrect read access. The variable in question is `a`. There is a whole bunch of other information available which, with some training, allows a programmer to find problems rather quickly.

There have been several attempts to extend the compiler to perform memory checking. `mudflap` has the advantage that its use does not change the ABI. The generated code can be mixed and matched with regularly compiled code. Specifically, it is possible to call functions in libraries; these do not have to be provided in special versions, usable by code compiled with `mudflap`. The instrumentation has to be paid for in performance. The bulk of the cost is in the `mudflap` runtime handling, the checking whether accesses are valid or not.

The behavior in case an error is detected and several other things can be controlled using the `MUDFLAP_OPTIONS` environment variable. To continue operation despite the error `-viol-nop` can be used. It is also possible to automatically start a debugger to attach to the program at the point of the error. This is enabled by using `-viol-gdb`. `mudflap` can print a list of leak memory blocks upon process termination if the `-print-leaks` option is added. There are a whole bunch of other option available, some to control functionality, others to tweak the inner workings of `mudflap`. Add the `-help` option to show the whole list.

Compared to `valgrind`, `mudflap` misses functionality to detect reads of uninitialized memory. Optimal results can only be achieved if almost all the code is compiled using `-fmudflap`. Otherwise one has to possibly sieve through false positive messages resulting from allocations the `mudflap` runtime has not seen.

4.2 Use Debug Mode

The C++ runtime library for gcc (libstdc++) has a special debug more in which case a lot of additional code is generated which performs tests at runtime. These tests will find a wide variety of problems which the C++ specification does not require to be noticed by default. The following little program shows the benefits:

```
#include <vector>
int main() {
  std::vector<int> v(10, 42);
  return v[11];
}
```

The variable `v` is defined as a vector with ten elements. The following instruction returns the 12^{th} element, which is out of range. Compiled normally the program runs and returns a more or less random value. Compiled with the macro `_GLIBCXX_DEBUG` defined (preferably on the command line of the compiler) the program will notice the error, print a message, and abort the program:

```
/usr/lib/gcc/x86_64-redhat-linux/4.0.0/../../../../
include/c++/4.0.0/debug/vector:192:
    error: attempt to subscript container with
    out-of-bounds index 11, but
    container only holds 10 elements.

Objects involved in the operation:
sequence "this" @ 0x0x7fffffb46fa0 {
  type = N15__gnu_debug_def6vectorIiSaIiEEE;
}
```

The option causes a slightly different template definition of the `vector` class to be seen by the compiler. The additional tests are added to the program code itself. These are not just calls to special functions in the C++ runtime which perform the additional checks. This makes the debug mode rather expensive and therefore it should not be enabled in production code.

4.3 Automatic Backtraces

If something goes wrong in a program when it is in production, the programmer needs to be able to find out in which state the program was before the crash. Otherwise the programmer would always have to reproduce the problem which might not always be feasible. Often the people running the code might not know how to describe the way to reproduce the program, it might take a long time, or it might be a non-deterministic event which caused the problem.

Therefore the goal should be to produce some output when the program crashes which the program user then can send along with the report of the incident to the programmer. This way the programmer can at least get a first impression where the problem was. In some cases this is already enough.

The easiest way to achieve this is to link the application with a little library called `libSegFault` (i.e., add `-lSegFault` to the command line when linking). This library registers, upon startup, a number of signal handlers for signals, which usually crash the application. The signals thus covered are `SIGSEGV`, `SIGILL`, `SIGBUS`,

SIGSTKFLT, SIGABRT, and SIGFPE. If such a signal arrives (and the program itself has not reset the signal handler) the code in libSegFault prints out quite a bit of information:

- a dump of the contents of the important registers
- a backtrace, which indicates the current code being executed and the function calls made to arrive at that position
- the map of all memory regions in use

After this information has been printed the program terminates as it would do without the signal handlers installed. If a program is not linked with libSegFault it still can benefit from it: all that is needed is that the program is started using the script catchsegv which comes with the GNU C library. This script adds the library to the program at runtime so that it can register the signal handlers and intercept fatal signals. The script does one more thing: if debug information is available for the program, it decodes the backtrace. Instead of addresses which the programmer would have to decode, the script will print file and function names, together with the exact line of the source code corresponding to the fatal instruction.

Internally libSegFault uses the functions declared in <execinfo.h>. The backtrace functions declared there help with the very machine-specific task of determining the call tree. Since the program can be in bad shape, these functions avoid calling complex code like malloc. Instead it is the programmer's responsibility to provide memory when it is needed. This can either be a statically allocated buffer, or some memory on the stack. These functions are usable by all programs, not just the code in libSegFault. This enables you to write crash diagnosis functions, which might be better suited for a specific program. The use is simple: the actual information is gathered with a call to backtrace() which fills a buffer provided by the caller.

The return value is the number of values filled into the buffer. This value, along with the array pointer, is then passed on to backtrace_symbols() or alternatively to backtrace_symbols_fd(). These functions try to provide as much information as possible about the addresses in the backtrace and create textual output. The difference is that the first function allocates an array of strings (the strings are also allocated) which contains the textual description. A pointer to the array is returned. This function might be dangerous to use in case of a program crash since the reason for the crash could very well be corrupted data in the malloc() implementation.

The backtrace_symbols_fd() function has no such problems. It writes the textual representation out to the file associated with the file descriptor passed to the functions as the third parameter.

A Locating Memory Allocations

The following definitions of macros which replace the allocator interfaces of the standard library help determining where a specific memory block was allocated. This provides a mean of determining where the memory block or any neighboring block. The code puts a string once before and once after the actually usable memory. The string consists of the location where the string was allocated (file name and line number) and the size of the memory block. In a debugger the place where the memory was allocated is thus visible.

```
#define malloc(sz) \
  ({ char __line[strlen(__FILE__) + 6 * sizeof(size_t) + 3]; \
     size_t __sz = sz; \
     int __n = sprintf(__line, "%c%s:%zu:%zu", \
                       '\0', __FILE__, __LINE__, __sz) + 1; \
     size_t __pre = roundup(__n, 2 * sizeof(size_t)); \
     char *__p = malloc(__sz + __pre + __n); \
     if (__p != NULL) \
       { \
         memset(mempcpy(__p, __line, __n - 1), ' ', __pre - __n); \
         __p[__pre - 1] = '\0'; \
         memcpy(__p + __pre + __sz, __line, __n); \
       } \
     (void *) (__p + __pre); })

#define calloc(szc, n) \
  ({ size_t __b = (szc) * (n); \
     char *__p = malloc(__b); \
     __p != NULL ? memset(__p, '\0', __b) : NULL; })

#define realloc(oldp, szr) \
  ({ char *__oldp = oldp; \
     char *__cp = __oldp; \
     while (*--__cp != ':'); \
     size_t __oldszr = atol(__cp + 1); \
     size_t __szr = szr; \
     char *__p = malloc(__szr); \
     if (__p != NULL) \
       { \
         memcpy(__p, __oldp, MIN(__oldszr, __szr)); \
         free(__oldp); \
       } \
     (void *) __p; })

#define free(p) \
  (void) ({ char *__p = (char *) (p) - 1; \
            while (*--__p != '\0'); \
            free(__p); })

#define valloc(szv) \
  ({ void *__p; \
     posix_memalign(&__p, getpagesize(), szv) ? NULL : __p; })

#define pvalloc(szp) \
  ({ void *__p; \
     size_t __ps = getpagesize(); \
     posix_memalign(&__p, __ps, roundup(szp, __ps)) ? NULL : __p; })

#define cfree(p) \
  free(p)
```

```
#define posix_memalign(pp, al, szm) \
  ({ size_t __szm = szm; \
     size_t __al = al; \
     char __line[strlen(__FILE__) + 6 * sizeof(size_t) + 3]; \
     int __n = sprintf(__line, "%c%s:%zu:%zu", \
                       '\0', __FILE__, __LINE__, __szm) + 1; \
     size_t __pre = roundup(__n, __al); \
     void *__p; \
     int __r = posix_memalign(&__p, __al, __pre + __szm + __n); \
     if (__r == 0) \
       { \
         memset(mempcpy(__p, __line, __n - 1), ' ', __pre - __n); \
         __p[__pre - 1] = '\0'; \
         memcpy((char *) __p + __pre + __szm, __line, __n); \
         *pp = (void *) ((char *) __p + __pre); \
       } \
     __r; })

#define memalign(al, sza) \
  ({ void *__p; \
     posix_memalign(&__p, al, sza) == 0 ? __p : NULL; })

#define strdup(s) \
  ({ const char *__s = s; \
     size_t __len = strlen(__s) + 1; \
     void *__p = malloc(len); \
     __p == NULL ? NULL : memcpy(__p, __s, __len); })
```

B References

[1] Ulrich Drepper, Red Hat, Inc., *How To Write Shared Libraries*, http://people.redhat.com/drepper/dsohowto.pdf, 2003.

[2] Ulrich Drepper, Red Hat, Inc., *Security Enhancements in Red Hat Enterprise Linux*, http://people.redhat.com/drepper/nonselsec.pdf, 2004.

[3] Frank Ch. Eigler, Red Hat, Inc., *Mudflap: Pointer Use Checking for C/C++*, http://gcc.fyxm.net/summit/2003/mudflap.pdf, 2003.

C Revision History

2004-12-12 First internal draft.

2005-2-19 Many fixes. Expand several sections.

2005-3-4 More language improvements. By Michael Behm.

2005-3-31 Yet more language improvements. By Michael Behm.

2005-4-12 And more. By Michael Behm.

2005-5-11 Some C++ stuff.

2009-3-30 Fix typo in getline example. By Martin Nagy.