Verification of CERT Secure Coding Rules: Case Studies

Syrine Tlili, XiaoChun Yang, Rachid Hadjidj, and Mourad Debbabi*

Computer Security Laboratory,
Concordia Institute for Information Systems Engineering,
Concordia University, Montreal (QC), Canada
{s_tlili,xc_yang,hadjidj,debbabi}@ciise.concordia.ca

Abstract. Growing security requirements for systems and applications have raised the stakes on software security verification techniques. Recently, model-checking is settling in the arena of software verification. It is effective in verifying high-level security properties related to software functionalities. In this paper, we present the experiments conducted with our security verification framework based on model-checking. We embedded a wide range of the CERT secure coding rules into our framework. Then, we verified real software packages against these rules for purpose of demonstrating the capability and the efficiency of our tool in detecting real errors.

1 Introduction

The `C` language is the language of choice for system programming accounting for its flexibility, portability, and performance. The `C` library provides programmers with a large set of functions that give them full control over memory management, file management, privilege management, etc. Nevertheless, security features in these functions are either absent or weak. As such, a secure usage of the `C` library functions fall to programmers responsibility. Unfortunately, skilled but inadvertent programmers often neglect security concerns in their implementation and produce code that may contain exploitable vulnerabilities.

For instance, some functions such as `gets()`, `sprintf()`, `strcpy()`, and `strcat()` are notoriously known to be vulnerable to buffer-overflow and denial of service attacks. These functions are now considered as deprecated and should never appear in a program. There are safe alternatives for these functions that should be used instead. However, a simple exchange of a deprecated function with its safe alternative is not enough to ensure security of programs. For instance, the `strncpy(char *s1, const char *s2, size_t n)` as opposed to its unsafe counterpart `strcpy()` takes as argument the number of bytes `n` to

* This research is the result of a fruitful collaboration between CSL (Computer Security Laboratory) of Concordia University, DRDC (Defence Research and Development Canada) Valcartier and Bell Canada under the NSERC DND Research Partnership Program.

R. Meersman, T. Dillon, P. Herrero (Eds.): OTM 2009, Part II, LNCS 5871, pp. 913–930, 2009.
© Springer-Verlag Berlin Heidelberg 2009

copy from the string referred to by `s2` to the array referred to by `s1`. Nevertheless, programmers must explicitly check in the code that `n` is less than the size of the destination array. Otherwise, `strncpy()` may overflow the destination array.

Despite the availability of a large range of secure coding books and websites, we still find many coding errors in software. The main reason is that programmers have stringent timing constraints to deliver their software. They cannot afford to spend their time resource in reading and learning secure coding practices. The main intent of this work is to demonstrate the efficiency and usability of our automated security verification tool [1] to automatically assess a set of real world `C` software. We also strive to help programmers in building secure software without the need to have high security skills and knowledge.

To this end, we specified an integrated within our tool a large set of the CERT secure coding rules. The latter are taken from the CERT website [2] that provides a valuable source of information to learn the best practices of `C`, `C++` and `Java` programming. It defines a standard that encompasses a set of rules and recommendations for building a secure code. Rules must be followed to prevent security flaws that may be exploitable, whereas recommendations are guidelines that help improve the system security. Notice that we target CERT rules that can be formally specified as finite-state automata and integrated in our tool. These automata-based rules represent a wide majority of the CERT standard.

The security verification tool that we use to conduct the experiments presented in this paper is based on the GCC compiler and the off-the-shelf model-checker for push-down systems, namely MOPED [3]. We choose to work with GCC to benefit from its language-independent intermediate representation GIMPLE [4] that facilitates the analysis of source code. Moreover, its multi-language support allows us to extend our framework to all languages that GCC compiles. The choice of model-checking technique is driven by the intent of covering a wide range of system-specific properties that we specify as finite state automata. The model-checker MOPED comes with a C-like modeling language called Remopla. We automatically serialize GIMPLE representation of `C` programs into Remopla models according to a given set of coding rule automata. The verification process detects a rule violation when its corresponding security automaton reaches a risky state.

The remainder of this paper is organized as follows: An informal overview of our approach and a description of the CERT secure coding rules are outlined in Section 2. We outline some advantages and challenges of using GIMPLE representation in Section 3. Our experiments are detailed in Section 4. We discuss the related work in Section 5 and we draw conclusion in Section 6.

2 Approach Overview

Figure 1 shows the architecture of our security verification environment. It carries out the verification process through different phases including security property specification, program model extraction, and property model-checking.

The first step of our security verification process requires the definition of a set of rules that describe the program secure behavior. Each rule is modeled as a

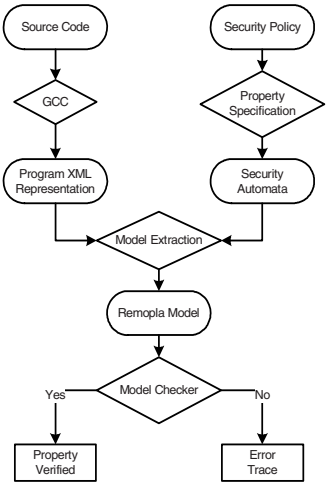


Fig. 1. Security verification framework

finite state automaton where the nodes define program states and the transitions match program actions. To ease the property specification, our tool supports syntactical pattern matching for program expressions and program statements. We use Remopla as a formal specification language of security automata. Then, the model extraction parses the GIMPLE representation of a source code in order to serialize it into Remopla model and combine it with the desired Remopla security automata.

The model-checking is the ultimate step of our process. The generated Remopla model is given as input to the Moped model-checker for security verification. The latter performs a flow-sensitive analysis that explores all execution paths of a program, though without pruning the unfeasible paths. An error is reported when a security automaton specified in the model, reaches a risky state. In that case, Moped generates an execution trace from the Remopla model that we map to a real trace in the verified `C` programs and we output to programmers. In the remainder of this paper, we illustrate how we used our tool to verify the secure coding rules from the CERT standard that we present hereafter.

2.1 CERT Coding Rules

To assist programmers in the verification of their code, we have integrated in our tool a set of secure coding rules defined in the CERT standard. As such, programmers can use our framework to evaluate the security of their code without the need to have high security expertise. Cert rules can mainly be classified into the following categories:

– *Deprecation rules*: These rules are related to the deprecation of legacy functions that are inherently vulnerable such as `gets` for user input, `tmpnam` for temporary file creation, and `rand` for random value generation. The presence

of these functions in the code should be flagged as a vulnerability. For instance, CERT rule `MSC30-C` states the following *Do not use the* `rand()` *function for generating pseudorandom numbers*

- *Temporal rules*: These rules are related to a sequence of program actions that appear in source code. For instance, the rule `MEM31-C` from the CERT entails to *Free dynamically allocated memory exactly once*. Consecutive free operations on a given memory location represents a security violation. Intuitively, these kind of rules are modeled as finite state automata where state transitions correspond to program actions. The final state of an automaton is the risky state that should never be reached.
- *Type-based rules*: These rules are related to the typing information of program expressions. For instance, the rule `EXP39-C` from the CERT states the following *Do not access a variable through a pointer of an incompatible type*. A type-based analysis can be used to track violations of these kind of rules.
- *Structural rules*: These rules are related to the structure of source code such as variable declarations, function inlining, macro invocation, etc. For instance, rule `DCL32-C` entails to *Guarantee that mutually visible identifiers are unique*. For instance, the first characters in variable identifiers should be different to prevent confusion and facilitates the code maintenance.

Our approach covers the two first categories of coding rules that we can formally model as finite state automata. In fact, we cover 31 rules out of 97 rules in the CERT standard. We also cover 21 recommendations that can be verified according to CERT.

3 Implementation

This section outlines details related to the implementation of our verification framework. We discuss some of the challenges and the benefits of using the GIMPLE intermediate representation of source code.

3.1 Macro Handling

The GIMPLE representation of programs is closely related to the environment under which the program is compiled. This tight coupling between the underneath environment and the considered code gives an appealing precision feature to our analysis compared to other approaches directly based on source code. Consider the following code snippet in Listing 1 taken from the `binutils-2.19.1` package. For code portability purposes, the macro `HAVE_MKSTEMP` is checked in `♯ifdef` to verify whether the system supports function `mkstemp()` for safe temporary file creation. If not, function `mktemp()` is used instead. A simplistic traversal of the source code would flag an error for the occurrence of `mktemp()` considered as an unsafe function for temporary file creation. Being based on GIMPLE representation, our analysis does not suffer this false alert. In fact, GIMPLE representation solves the conditional `♯ifdef`, and one of the two temporary file

functions will appear in the GIMPLE code with regard to the compilation environment. In our case, the machine used to conduct the experiments supports `mkstemp()` which is present in the GIMPLE code of Listing 2

Listing 1. Sample `C` code from `binutils-2.19.1` with macros

```
#ifdef HAVE_MKSTEMP
    fd = mkstemp (tmpname);
#else
    tmpname = mktemp (tmpname);
    if (tmpname == NULL)
        return NULL;
    fd = open (tmpname, O_RDWR | O_CREAT | O_EXCL, 0600);
#endif
```

Listing 2. GIMPLE representation of code in Listing 1

```
D.8401 = mkstemp (tmpname);
fd = D.8401;
if (fd == -1)
    { D.8402 = 0B;
      return D.8402;
    }
```

This points out the important fact that the verification of software should be performed on the same environment intended for their real usage. Besides, the verification should be performed on hostile environments to predict as much worst execution scenarios as possible.

3.2 Temporary Variables

The GIMPLE representation breaks down program expressions into SSA form in which each variable is defined exactly once [4]. This form of representation involves the definition of temporary variables that hold intermediate values. Consider the call to `malloc` function in Listing 3, its corresponding GIMPLE code in Listing 4 splits the `malloc` call into two sub-expressions involving a temporary variable `D.1861`.

Listing 3. Sample `C` with memory allocation

```
p = malloc(BUFSIZ)
if (!p)
    return -1;
...
free(p);
return 1;
```

The return value of `malloc()` is assigned to a temporary variable `D.1861`. Then, the latter is cast and assigned to pointer `p`. The usage of temporary variables presents a challenge for pattern matching. In this example, variable `D.1861` matches the pattern for the return value of `malloc()`, whereas variable `p` matches the pattern for the call to `free()` argument.

Listing 4. GIMPLE representation of code in Listing 3

```
D.1860 = malloc(5);
p = (char *) D.1860;
if ( p == 0B) {
    D.1861 = -1;
    return D.1861;
}
...
free(p);
D.1861 = 1;
return D.1861;
```

Without considering relations between temporary variables, the verification process flags an erroneous warning for the deallocation of an uninitialized pointer. The expressiveness of the GIMPLE representation helped us to overcome this challenge. In fact, GIMPLE keeps track of the original definition of temporary variables. In the given example, we are able to recognize that temporary variable `D.1861` is an intermediate representation of `p` and avoid spurious warnings.

4 Experimentation

In this section, we detail our conducted experiments that consist in verifying a set of well-known and widely used open-source software against a set of CERT secure coding rules. We strive to cover different kinds of security coding errors that skilled programmers may inadvertently produce in their code. In the sequel of this section, we detail the results of the experimentation that we conducted on large scale `C` software. The content of the tables that present the experimentation results is described in the following paragraph. The three first columns define the package name, the size of the package, and the program that contains coding errors. The number of reported error traces is given in the fifth column (Reported Errors). After manual inspection of the reported traces, we classify them into the three following columns: column (Err) for potential errors, column (FP) for false positive alerts, and column (DN) for traces that are complicated and time consuming for manual inspection. The checking time of programs is given in the last column.

4.1 Unsafe Environment Variables

CERT Coding Rules:

- `STR31-C`: Guarantee that storage for strings has sufficient space for character data and the null terminator.
- `STR32-C`: Null-terminate byte strings as required.
- `ENV31-C`: Do not rely on an environment pointer following an operation that may invalidate it

String manipulation in `C` programming is famous for spawning exploitable errors in source code such as inappropriate format string, buffer overflows, string truncations, and not null-terminated strings. For our experiments, we focus on the following CERT rules:

- Rule `STR31-C` disciplines the usage of string copy functions to prevent buffer overflows and truncation errors that arise from copying a string to a buffer that is not large enough to hold it.
- Rule `STR32-C` stresses on the need of a null character to mark the end of a string. For flexibility sake, the `C` language does not limit string sizes and depends on the presence of a null character `\0` to mark the end of a string. The absence of this character results in buffer overflows and denial of service attacks.
- Rule `ENV31-C` targets the safe usage of environment functions to prevent bad assumption resulting from inconsistent environment values.

The risk of string errors increases even more when using string pointers to environment variables. In fact, programs' execution environment should never be trusted and should be considered as hostile to safe execution. From this conservative assumption, all values requested from the environment should be checked before usage: null pointer checks, bound checks, and null-termination checks. The `C` library contains a set of environment functions that are widely used despite their notorious reputation of being unsafe. Among these functions, we have `ttyname()` and `getenv()`. These functions return a string with unknown size that may be not null-terminated. On failure, these functions return a null pointer. Besides, these functions are not reentrant. In other words, if multiple instances of the same function are concurrently running, it may lead to inconsistent states. Attackers may take advantage of this reentrant characteristic to invalidate the values of environment variables. The CERT rule `ENV31-C` targets the safe usage of environment functions to prevent bad assumption resulting from inconsistent environment values. We define the automaton in Figure 2 to detect the unsafe usage of environment functions. Table 1 illustrates the results of our experimentation for a given set of software. The fifth column indicates the reported error traces. After inspecting the traces, we distinguish false positives from what we believe to be a potential error in the sixth column. We discuss in the following paragraphs some of the reported errors.

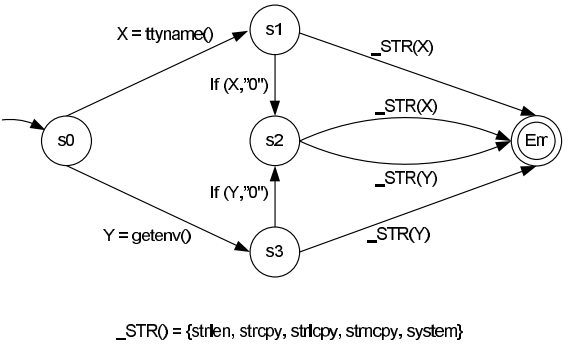


Fig. 2. Environment function automaton

Table 1. Unsafe Environment Variables

Package	LOC	Program	Rule	Reported Errors	Err	Checking time (Sec)
openssh-5.0p1	58K	sshd	STR31-C	1	0	0.15
krb5-1.6	276K	kshd	ENV31-C	2	2	0.33
		kshd	STR32-C	2	2	0.33
patchutils-0.1.5	1.3K	interdiff	STR32-C	1	1	0.06
kstart-3.14	4.4K	krenew	STR32-C	1	1	0.06
inetutils-1.6	276 K	tftp	STR31-C	1	1	0.54
		telnet	STR31-C	1	1	0.52
chkconfig-1.3.30c	4.46 K	chkconfig	STR32-C	1	1	0.52
freeradius-2.1.3	77 K	radiusd	STR32-C	1	1	0.52

The code in Listing 5 is taken from program `sshd` of `openssh-5.0p1`. It triggers a warning when analyzed with our tool. In fact, the return value `name` of `ttyname()` is copied using the function `strlcpy()`. This function ensures the null-termination of the destination buffer `namebuf` provided that `namebuflen` is properly set. If the size of `name` is bigger than `namebuflen`, then there is a possible string truncation error as mentioned in the programmers comments. From their comments, we assume that programmers intentionally did not handle the possible string truncation as they do not consider it as an exploitable error. We consider this error trace as a false positive.

Listing 5. Unsafe usage of `ttyname()` in `openssh-5.0p1` (Rule `STR31-C`)

```
name = ttyname(*ttyfd);
if (!name)
    fatal("openpty returns device for which ttyname fails.");
strlcpy(namebuf, name, namebuflen); /* possible truncation */
return 1;
```

The code fragment of Listing 6 is taken from `krb5-1.6`. It is a good example to show what not to do when using environment variables. It calls `getenv()` to get the value of environment variable `KRB5CCNAME`.

Listing 6. Unsafe usage of `getenv()` in `krb5-1.6` (Rules `ENV31-C` and `STR32-C`)

```
if (getenv("KRB5CCNAME")) {
 int i;
 char *buf2 = (char *)malloc(strlen(getenv("KRB5CCNAME"))
                             +strlen("KRB5CCNAME=")+1);
 if (buf2) {
    sprintf(buf2, "KRB5CCNAME=\%s",getenv("KRB5CCNAME"));
 ...
}
```

In this code, `getenv()` is called three consecutive times. There is absolutely no guarantee that these three calls return the same value. An attacker may take advantage of the time race between each call to modify the value of variable `KRB5CCNAME`.

– Between the first and the second call, an attacker can remove variable `KRB5CCNAME` from the environment and the second call to `getenv()` returns a null pointer. In that case, function `strlen()` would have a null argument and would generate a segmentation fault.
– Besides, `getenv()` is used a third time as an argument to `sprintf()` which is vulnerable to buffer overflow and should be avoided according to CERT rule `FIO33-C`. We assume that the allocation of `buf2` is successful. Between the second call and the third call to `getenv()`. An attacker may change the value of `KRB5CCNAME` and set it to a larger string than the one considered for the memory allocation. The call to `sprintf()` is then prone to overflow the memory space of `buf2`.

We definitely consider this piece code as unsafe since it makes bad assumptions on nasty values of environment variables.

4.2 Unchecked Return Values

CERT Coding Rules:

– `MEM32-C`: Detect and handle memory allocation errors.
– `EXP34-C`: Ensure a null pointer is not dereferenced

Unfortunately, programmers very often omit to handle erroneous return values from function calls. They make wrong assumptions on the successful termination of callee functions. According to the Coverity scan report, the use of unchecked return values represents 25% of programming errors [5]. Error handling omission can lead to system crashes especially for memory allocation functions that return null pointer on failure. Therefore, rule `MEM32-C` entails that the return value of memory allocation functions should be checked before being used to prevent the nasty dereference of null pointers. Besides, rule `EXP34-C` emphasizes that null pointers should not be dereferenced. Table 2 illustrates the analysis results of the security automaton depicted in Figure 3.

We reviewed the reported error traces and mark them all as real errors. They contain a allocation operation that is never followed by a null check of the returned pointer. We give in Listing 7 a code snippet from the `apache-1.3.41` that uses the return pointer of `malloc()` without null check.

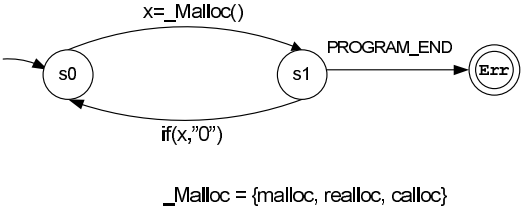


Fig. 3. Null check automaton

Table 2. Return value checking

Package	LOC	Program	Reported Errors	Checking time (Sec)
amanda-2.5.1p2	87K	chg-scsi	1	28.87
apache-1.3.41	75K	ab	1	0.4
bintuils-2.19.1	986K	ar	1	0.74
freeradius-2.1.3	77K	radeapclient	1	1.06
httpd-2.2.8	210K	ab	1	0.5
openca-tools-1.1.0	59K	openca-scep	2	2.6
shadow-4.1.2.2	22.7K	groupmems	1	3.08
		groups	1	2.81
		usermod	1	2.82
		id	1	2.80
		useradd	1	2.81
		vipw	1	3.05
zebra-0.95a	142K	ospf6test	1	15.13

Listing 7. Use with null-check in `apache-1.3.41`

```
con = malloc(concurrency * sizeof(struct connection));
memset(con, 0, concurrency * sizeof(struct connection));
```

4.3 Race Conditions

CERT Coding Rules:

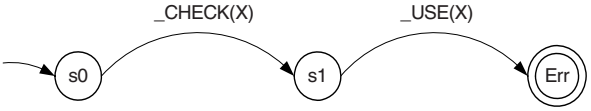
- `POS35-C` Avoid race conditions while checking for the existence of a symbolic link.
- `FIO01-C` Be careful using functions that use file names for identification.

The Time-Of-Check-To-Time-Of-Use vulnerabilities (TOCTTOU) in file accesses [6] are a classical form of race conditions. In fact, there is a time gap between the file permission check and the actual access to the file that can be maliciously exploited to redirect the access operation to another file. Figure 4 illustrates the automaton for race condition detection. It flags a check function followed by a subsequent use function as a TOCTTOU error. The analysis results are given in Table 3.

Listing 8 illustrates a race condition error in package `zebra-0.95a`. The `stat()` function is called on file `fullpath_sav` before being accessed by calling function `open()`. Being based on pathname instead of file descriptor renders these functions vulnerable to TOCTTOU attacks as detailed in [6].

Listing 8. File race condition in `zebra-0.95a`

```
if (stat (fullpath_sav, &buf) == -1) {
    free (fullpath_sav);
    return NULL;
}
...
sav = open (fullpath_sav, O_RDONLY);
...
while(( c = read (sav, buffer, 512)) > 0)
...
```



_CHECK = access, stat, statfs, statvfs, lstat, readlink, tempnam, tmpnam, tmpnam_r

_USE = acct, au_to_path, basename, catopen, chdir, chmod, chown, chroot, copylist, creat, db_initialize, dbm_open, dbminit, dirname, dlopen, execl, execle, execlp, execv, execve, execvp, fattach, fdetach, fopen, freopen, ftok, ftw, getattr, krb_recvauth, krb_set_tkt_string, kvm_open, lchown, link, mkdir, mkdirp, mknod, mount, nftw, nis_getservlist, nis_mkdir, nis_ping, nis_rmdir, nlist, open, opendir, pathconf, pathfind, realpath, remove, rename, rmdir, rmdirp, scandir, symlink, system, t_open, truncate, umount, unlink, utime, utimes, utmpname

Fig. 4. Race condition of file access (TOCTTOU)

Table 3. File race condition TOCTTOU

Package	LOC	Program	Reported Errors	Err	FP	DN	Checking time (Sec)
amanda-2.5.1p2	87K	chunker	1	0	1	0	71.6
		chg-scsi	3	2	1	0	119.99
		amflush	1	0	0	1	72.97
		amtrmidx	1	1	0	0	70.21
		taper	3	2	1	0	84.603
		amfetchdump	4	1	0	3	122.95
		driver	1	0	1	0	103.16
		sendsize	3	3	0	0	22.67
		amindexd	1	1	0	0	92.03
at-3.1.10	2.5K	atd	4	3	1	0	1.16
		at	4	3	1	0	1.12
bintuils-2.19.1	986K	ranlib	1	1	0	0	2.89
		strip-new	2	0	1	0	5.49
		readelf	1	1	0	0	0.23
freeradius-2.1.3	77K	radwho	1	1	0	0	1.29
openSSH-5.0p1	58K	ssh-agent	2	0	0	2	22.46
		ssh	1	0	1	0	100.6
		sshd	6	3	1	2	486.02
		ssh-keygen	4	4	0	0	87.28
		scp	3	2	0	1	87.95
shadow-4.1.2.2	22.7K	usermod	3	1	0	2	9.79
		useradd	1	1	0	0	11.45
		vipw	2	2	0	0	10.32
		newusers	1	1	0	0	9.2
zebra-0.95a	142K	ripd	1	1	0	0	0.46

Listing 9 contains a sample code extracted from package `amanda-2.5.1p2`. The `mkholdingdir()` function is used inside a loop. Our tool goes through the loop and considers that there is a path where `stat(diskdir,...)` is a check function and `mkdir(diskdir,...)` is a use function that corresponds to the pattern of TOCTTOU errors. We actually consider this reported error as a false positive since there are paths where the `mkdir()` call does not depend on the result of the `stat()` check. Besides, the return value of the `mkdir()` is used to check the successful creation of the directory.

Listing 9. False positive TOCTTOU in `amanda-2.5.1p2`

```
while (db->split_size > (off_t)0
    && dumpsize >= db->split_size)
{
    ...
    mkholdingdir(tmp_filename);
    ...
}
mkholdingdir(char *  diskdir){
    struct stat stat_hdp;
    int success = 1;
    ...
    else if (mkdir(diskdir, 0770) != 0 && errno != EEXIST)
    {...}
    else if (stat(diskdir, &stat_hdp) == -1)
    { ...
```

4.4 Unsafe Temporary File Creation

CERT Coding Rule:

– `FIO43-C`: Do not create temporary files in shared directories.

Very often software applications create and maintain temporary files for different purposes such as information sharing, temporary data storing, and computation speeding up. Usually applications store temporary files in shared folders, then terminate execution and leave these files behind. This bad management of temporary files exposes private and sensitive data and offers to attackers the possibility to hijack temporary files and tamper with their content. The impact of such attacks is very high especially when these targeted files are set with high privileges. Therefore, programmers must properly create, protect, and delete temporary files. The standard `C` library provides a set of functions for temporary file creation. However, most of these functions are vulnerable to various forms of attacks and must be used with precaution. We detail in the following paragraphs the temporary file discipline entailed by the CERT rule `FIO43-C` and modeled in automaton of Figure 5. Table 4 gives the verification results for a set of packages against the security automata of rule `FIO43-C`.

– Temporary file creation: A temporary file must have a unique name to avoid collisions with existing files. The `C` functions `tmpnam()`, `tempnam()`, `tmpfile()`, and `mktemp()` generate a unique file name when invoked. However, these functions suffer a race condition between the file name generation

and the file creation that can be exploited by attackers. We refer to this error as `FIO43-C-1` in Table 4.

– Setting appropriate permissions: Since temporary files are usually created in shared folders, it is highly required to set appropriate permissions to these files to ensure their protection against attackers. As such, a call to `umask(077)` must be done before a call to `mkstemp` to limit the permissions of the resulting temporary file to only the owner. We refer to this error as `FIO43-C-2` in Table 4.

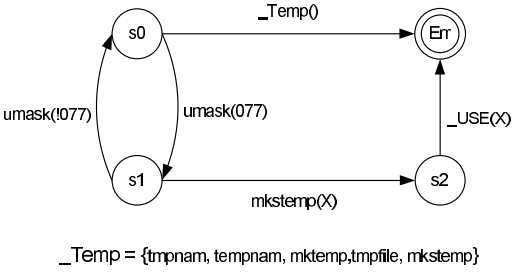


Fig. 5. Temporary file security automaton

Table 4. Temporary file errors

Package	LOC	Program	Reported Errors	Checking time (Sec)	CERT Rule
openssh-5.0p1	58K	ssh-keygen	1	9.21	`FIO43-C-2`
		sshd	1	50.6	`FIO43-C-3`
		ssh-rand-helper	1	7.52	`FIO43-C-3`
apache-1.3.41	75K	htpasswd	1	0.13	`FIO43-C-1`
		htdigest	1	0.09	`FIO43-C-1`
shadow-4.1.1	22.7K	useradd	1	2.75	`FIO43-C-2`
patchutils-0.1.5	1.3K	interdiff	3	0.17	`FIO43-C-{1,2}`
		filterdiff	1	0.11	`FIO43-C-1`
krb5-1.6	276K	kprop	1	0.11	`FIO43-C-1`
kstart-3.14	4.4K	k4start	1	0.14	`FIO43-C-2`
		k5start	1	0.15	`FIO43-C-2`
		krenew	1	0.11	`FIO43-C-2`
chkconfig-1.3.30c	4.46K	chkconfig	1	0.11	`FIO43-C-1`
inn-2.4.6	89K	nntpget	1	0.32	`FIO43-C-2`
		shrinkfile	1	0.27	`FIO43-C-2`
		innxmit	1	0.52	`FIO43-C-2`
		makehistory	2	0.37	`FIO43-C-2`
binutils-2.19.1	986K	ranlib	1	1.9	`FIO43-C-2`
emacs-22.3	986K	update-game-score	1	0.19	`FIO43-C-3`

– Race conditions: Functions that create temporary files are considered as check functions, as defined in Section 4.3, that are subject to race condition errors when their filename argument is used in a subsequent system call. We refer to this error as `FIO43-C-3` in Table 4.

The sample code in Listing 10 is taken from `make-3.81` package. The GIMPLE representation of that code is given in Listing 11. This code is quite similar to the code fragment in Listing 1. Both codes use the `♯ifdef` macro to verify the system support of function `mkstemp()`. Otherwise, the system uses `mktemp()`. Checking for system supports of safe functions is a good practice for secure programming. However, this fragment is not error free. Suppose that `mkstemp()` is used, its file name argument should never appear in any subsequent system call according to the CERT rule `FIO43-C`. Hence, the call to `fopen()` with the same file name presents a file race condition error detailed in Section 4.3.

Listing 10. Temporary file error in `emacs-22.3`

```
#ifdef HAVE_MKSTEMP
    if (mkstemp (tempfile) < 0
#else if (mktemp (tempfile) != tempfile
#endif
    || !(f = fopen (tempfile, "w")))
return -1;
```

Listing 11. GIMPLE representation of source code in Listing 10

```
D.4565 = mkstemp (tempfile);
if (D.4565 < 0) { goto <D4563>; }
/*...*/
D.4566 = fopen (tempfile, &"w"[0]);
f = D.4566;
```

In Listing 1, `fopen` is called only when `mktemp()` is used for the temporary file creation. The `O_EXCL` flag provides an exclusive access to the file to prevent unauthorized access. The error that we trigger for this code is related to non usage of the `umask(077)` call to set the temporary file permissions.

4.5 Use of Deprecated Functions

CERT Coding Rules:

– `FIO33-C`: Detect and handle input output errors resulting in undefined behavior.
– `POS33-C`: Do not use `vfork()`.
– `MSC30-C`: Do not use the `rand()` function for generating pseudorandom numbers

The CERT coding rules forbid the usage of deprecated `C` functions as they are readily vulnerable to attacks such buffer overflows, code injection, and privilege

escalation. The usage of safe alternatives is required as a preventive measure. We present hereafter the set of CERT rules that we verify with our tool:

- Rule `MSC30-C` for random number generation: The `rand()` function produces numbers that can easily be guessed by attackers and should never be used especially for cryptographic purposes. The CERT recommends using function `random()` instead.
- Rule `POS33-C` for process management: The `vfork()` function suffers race conditions and denial of service vulnerabilities and should never be used. Programmers should consider the usage of `fork()` as a safe alternative.
- Rule `FIO33-C` for string manipulation: The CERT deprecates the usage of function `gets()`, `sprintf()`, and `vsprintf()` since they are extremely vulnerable to buffer overflow attacks. Microsoft developed safe alternatives to `C` string functions [7] that are recommended by the CERT standard.

Figure 6 depicts the automaton for the detection of deprecated functions. From the analysis results in Table 5, we deduce that deprecated functions as still used in many software. For instance, function `rand()` is used in package `apache-1.3.41` for password generation as illustrated in Listing 12. We believe that it should be replaced with a safe alternative to provide an adequate level of password security.

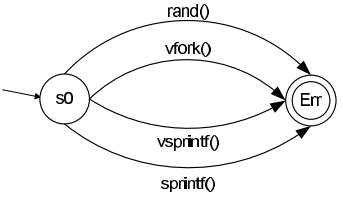


Fig. 6. Deprecated function automata

Table 5. Usage of deprecated functions

Package	LOC	Program	Rule	Reported Erros	Err	Checking time (Sec)
apache-1.3.41	75 K	htpasswd	`MSC30-C`	2	2	0.25
inetutils-1.6	276 K	rcp	`POS33-C`	1	1	0.47
krb5-1.6	276 K	rcp	`POS33-C`	1	1	0.08
		kshd	`FIO33-C`	many	many	0.20
zebra-0.95a	142 K	ripd	`MSC30-C`	1	0	0.17
emacs-22.3	242 K	update-game -score	`MSC30-C`	1	0	0.30
wget-1.11.4	24.5 K	wget	`FIO33-C`	many	many	0.20
chkconfig-1.3.30c	4.46 K	chkconfig	`FIO33-C`	many	many	0.34

Listing 12. Unsafe usage of `rand()` for password generation in `apache-1.3.41`

```
static int mkrecord(char *user, char *record, size_t rlen,
 char *passwd, int alg) {
    /*...*/
    switch (alg) {
     /*...*/
     case ALG_APMD5:
         (void) srand((int) time((time_t *) NULL));
         ap_to64(&salt[0], rand(), 8);
         salt[8] = '\0';
     /*...*/
```

Listing 13. Using `rand()` to compute time jitter in `zebra-0.95`

```
rip_update_jitter (unsigned long time){
  return ((rand () % (time + 1)) - (time / 2));
}
void rip_event (enum rip_event event, int sock){
 /*...*/
 jitter = rip_update_jitter (rip->update_time);
 /*...*/
}
```

In the case of packages `zebra-0.95a` and `emacs-22.3`, `rand()` is used for time synchronization purposes. Listing 13 shows the usage of `rand()` in the routing package `zebra-0.95` to compute a time jitter. We do not know whether the timing for these programs are security relevant and cannot claim that the use of `rand()` is an exploitable error.

5 Related Work

This section presents approaches and tools based on static analysis and model-checking for vulnerability detection in source code.

MOPS is a pushdown model-checking tool for `C` programs [8]. It provides an automata-based language for the definition of temporal security properties. It has been successful in detecting programming errors in the Linux kernel. Notice that MOPS has been designed and implemented for exclusively handling `C` language. Our approach benefits from the GIMPLE representation in order to be extended to all languages that GCC compiles. Moreover, the integration of the CERT secure coding rules into our tool renders it more practical for programmers that often do not know what errors to detect and to fix.

MetaCompilation (MC) is a static analysis tool that uses a flow-based analysis approach for detecting temporal security errors in `C` code [9]. With the MC approach, programmers define their temporal security properties as automata written in a high-level language called Metal [10] based on syntactic pattern matching. In our approach, we benefit from the expressiveness of the procedural Remopla language to achieve the same level of expressiveness of Metal. A key difference is that metal patterns reference the source code directly, whereas our patterns are closer to the compiler representation and reference GIMPLE constructs. Soundness is another important difference between our approach and

MC approach. Our analysis is sound with respect to generated program model, whereas MC sacrifices soundness for the sake of scalability.

BLAST [11], SAT [12] and SLAM [13] are data-flow sensitive model-checkers based on predicate abstraction. They use an iterative refinement process to locate security violations in source code. Both are mainly used to verify small software of device drivers. Despite the precision of their approach, their iterative process introduces the risk of non-termination and does not scale to large software.

ITS4 [14] and RATS [15] perform lexical analysis of source code to detect vulnerabilities. Unfortunately, these tools perform in a flow-insensitive and alias insensitive manner rendering them prone to very high rates of spurious warnings.

6 Conclusion

In this paper, we have detailed the experiments on large scale `C` software conducted with our security verification tool [1]. The experimentation results demonstrate the efficiency and the usability of our tool in detecting real errors in real-software packages. We list hereafter the appealing characteristics of our automatic security tool:

- Practical: The integration of the CERT secure coding rules within our tool renders it practical and valuable for assisting programmers in building secure software.
- Precision: As shown in this paper, environment information that is not captured in source code is revealed in the GIMPLE representation. As such, our analysis have more insight on the environment under which a software should execute.
- Flexible: Being based on the language-independent GIMPLE representation, our approach has the potential to be extended to all languages that GCC supports.

References

1. Hadjidj, R., Yang, X., Tlili, S., Debbabi, M.: Model-checking for software vulnerabilities detection with multi-language support. In: PST 2008: Proceedings of the 2008 Sixth Annual Conference on Privacy, Security and Trust, pp. 133–142. IEEE Computer Society, Washington (2008)
2. CERT Secure Coding Standards (April 2009), `http://www.securecoding.cert.org`
3. Reps, T., Schwoon, S., Jha, S., Melski, D.: Weighted pushdown systems and their application to interprocedural dataflow analysis. Science of Computer Programming 58(1–2), 206–263 (2005); Special Issue on the Static Analysis Symposium 2003
4. Novillo, D.: Tree-SSA: A New Optimization Infrastructure for GCC. In: Proceedings of the GCC Developers Summit3, Ottawa, Ontario, Canada, pp. 181–193 (2003)
5. Coverity: Coverity Prevent for C and C++, `http://www.coverity.com/main.html`

6. Bishop, M., Dilger, M.: Checking for Race Conditions in File Accesses. Computing Systems 2(2), 131–152 (1996)
7. Specification for safer, more secure c library functions. Technical Report tech. report ISO/IEC TR 24731, Int'l Organization for Standardization (September 2005), http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1135.pdf
8. Chen, H., Wagner, D.A.: MOPS: an Infrastructure for Examining Security Properties of Software. Technical Report UCB/CSD-02-1197, EECS Department, University of California, Berkeley (2002)
9. Ashcraft, K., Engler, D.: Using Programmer-Written Compiler Extensions to Catch Security Holes. In: SP 2002: Proceedings of the 2002 IEEE Symposium on Security and Privacy, pp. 143–159. IEEE Computer Society, Washington (2002)
10. Hallem, S., Chelf, B., Xie, Y., Engler, D.: A System and Language for Building System-Specific, Static Analyses. In: PLDI 2002: Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation, pp. 69–82. ACM, New York (2002)
11. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy Abstraction. In: POPL 2002: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages, pp. 58–70. ACM, New York (2002)
12. Clarke, E., Kroening, D., Sharygina, N., Yorav, K.: Predicate Abstraction of ANSI-C Programs Using SAT. Formal Methods in System Design 25(2-3), 105–127 (2004)
13. Ball, T., Rajamani, S.K.: The slam project: debugging system software via static analysis. SIGPLAN Not. 37(1), 1–3 (2002)
14. Viega, J., Bloch, J.T., Kohno, Y., McGraw, G.: ITS4: A Static Vulnerability Scanner for C and C++ code. In: ACSAC 2000: Proceedings of the 16th Annual Computer Security Applications Conference, p. 257. IEEE Computer Society, Los Alamitos (2000)
15. Fortify Software. Rats - rough auditing tool for security (April 2009), http://www.fortify.com/security-resources/rats.jsp