Proactive security measures in coding

Ville Valkonen

University of Tampere
School of Information Sciences/CS
Master's Thesis
Supervisor: Heikki Hyyrö
7.12.2014

University of Tampere
School of Information Sciences/CS
Ville Valkonen: Proactive security measures in coding
Master's Thesis, 52 pages; Appendices, 19 pages
December 2014

Abstract

There are several ways to mitigate security breaches proactively. This thesis introduces portable security methods that can be adapted in any Unix-like operating system. These methods can help to mitigate the harm done by a malicious attacker who has already gained a partial access into the system. The main focus in the thesis is to give an idea how attacks can be pursued and how to protect against them.

The first research question is: What proactive steps can be done to reduce errors and vulnerabilities in code before it is released? What methods can be adapted to harden the code and make it less penetrable? I examine a few design principles which are known to be good against malicious activities.

The second research question is: What is the state of the static analyzers in modern compilers, when compared to dedicated static analyzers? This a part of the thesis introduces automatic ways to check code against unsafe API or system call usages. Static code analysis has been around for awhile and performed heuristics of modern analyzers are highly sophisticated. Freely available open source analyzers are tested against example flaws and the results are reviewed. In the last section, analyzers are tested against a real world program which are used widely.

As a conclusion, many of the shown proactive security measures will help to mitigate against malicious activity, as proven by the real world code analysis.

Keywords and terms: software security, static code analysis, design patterns, code review, proactive security.

Table of contents

1 Introduction 1
2 Research questions 2
3 Preliminares 3
3.1 Memory 4
3.2 Input validation problem 6
3.3 Buffer overflow 7
3.4 SQL injections 8
4 Proactive methods 10
4.1 Design principles 10
4.2 Compiler and language based safety methods 12
4.3 Protecting the memory 16
4.4 Access control methods 18
5 Static code analysis 20
5.1 Types of checks 20
5.2 Annotations 22
5.3 Control flow and data flow analysis 23
5.4 Survey of analyzers 23
6 Test environment setup 29
6.1 Operating system 29
6.2 Compilers setup 30
6.3 Predefined flaws 32
7 Analyzing predefined flaws results 37
7.1 Compilers (Clang and GCC) 37
7.2 Cppcheck 37
7.3 Flawfinder 38
7.4 Scan-build 38
7.5 Splint 41
7.6 Conclusion of predefined analyzing results 41
8 A real world example 44
8.1 Compilers (Clang and GCC) 45
8.2 Cppcheck 46
8.3 Flawfinder 46
8.4 Scan-build 46
8.5 Splint 47
8.6 Manual code review 48

8.7 Conclusion of the real world example 49
9 Conclusion and future studies 51
References 53
Appendix A Java JDBC Stack trace 59
Appendix B example_flaws.c 60
Appendix C Makefile 67
Appendix D codechecks.log 69
Appendix E Makefile of tvheadend (patch) 76
Appendix F splint.sh 78

1 Introduction

This thesis examines different proactive security methods that can be integrated into programming development processes in order to reduce the damage made by various malicious activities. Some of the methods are easier to adopt in general programming guidelines, while others are more dependent on the environment. The main focus of this thesis is on Unix-like operating systems.

The difference between highly secure systems and non-secure systems can be described in the following way: non-secure systems concentrate on implementing certain functionality, while secure system's main goal is to achieve functionality in a safe manner. Designing a secure system is an arduous process and difficult to get correct. Basically, a system should *fail safely*, should run the *least privilege* and should not be too complicated to understand (*Keep It Simple Stupid*, KISS [The Free Dictionary, 2012]). The thesis will walk-through how design patterns and ideologies harden the program and make them more robust.

The thesis will also examine techniques that static code analyzers use, explains the theory behind them, and presents through examples how analyzers can reveal bugs in software. Analyzers can be integrated to be a part of the designing process, and triggered automatically after every successful build or release. If the result is being examined thoroughly, it will inevitably eliminate easy programming errors and security fiascos from the final product. Aforementioned topics are discussed in greater detail below.

When studying and implementing secure systems, one should member what Erik Poll [Poll E., 2011a] has stated: *The attacker only has to get lucky once, the defender has to get it right all the time.*

2 Research questions

My first research question is: What proactive steps can be done to reduce errors and vulnerabilities in code before it is released? Simple bugs can cause financial loss, power outage to an entire city or even death. Even if precautions would have been carefully done, there is no guarantee of absolute bug free software. Someone could ask “why bother to use extra checks if there is no guarantee that it would prevent failures? Extra checks take time from coding and costs money!”

I can start answering that question through a simple well known case. Ariane 5 rocket was launched on its maiden cruise and less than five minutes from the launch the space shuttle went to a self destruction mode and exploded. What happened? A computer program tried to fit a 64 bit number into a 16 bit space [Gleick J., 2010], causing an *integer overflow*. Could it have been avoided by using static code analysis? It is easy to speculate retrospectively. How about other bugs, such as a web *daemon* that leaks an anonymous user a super user rights, when CGI-script execution fails?

My second research question is: Are modern compiler integrated checks sufficient for day to day use and how do they perform, when compared against static analyzers?

I am participating in a few open source projects and also write code on my free time. I have used static code analysis as a part of my coding procedures for awhile and found it useful to catch obvious bugs before the software release and even afterwards. By doing comparisons between different analyzers I hope to gain more understanding of each tool’s strengths and weaknessess and learn to use the most suitable tool for a specific task.

The thesis tries to answer the research questions by explaining simply the theories that are widely acknowledged and approved by the security field, and testing some of the theories in practice.

3 Preliminares

This section focuses on different technologies and methods that are widely adopted by many known security environments, programs and security focused operating systems. Hence, systems that lack an exhaustive testing process or are in a prototype state are excluded from this thesis. The thesis starts with the basics that must be known before one can proceed. This section also presents shortcomings that current approaches might have. This thesis tries to give a general understanding of how vulnerabilities can be pursued but does not try to cover all possible fields or methods. There are fields regarding concurrent processing, networking and input/output processing that are omitted in this thesis, though some of them might be mentioned briefly.

I start by defining a *software flaw* and what proactive actions can be done to prevent or reduce the damage. The software flaw is an unexpected operation of a program. It can expose the system to a great danger, leak confidential information or crash the whole system. On some cases, especially in network related environments, the software flaw can expose a huge number of computers to a danger. If it is possible to gain more information from the system than it is designed to offer, or when the system is set up to work in a way it was not intended, the system can be seen as vulnerable.

There are several different types of vulnerabilities. One can measure vulnerabilities according to their severity, e.g., by measuring their level of criticality. Security companies rank vulnerabilities in a different way but certain parts of the rankings are the same. One of the unarguable ranking categories is the type of domain, *a local exploit* and *a remote exploit*. A local exploit demands that an attacker must have local access into the system. The local access can be physical or non-physical (a user account in the machine). Instead, the remote exploit does not have this demand, and it can therefore be seen as more dangerous. Also, remote exploits are considered more risky, since there is no need to have an account in the machine. Hence, vulnerabilities can be exploited via Internet.

Communication in the Internet is based on packet data. Thus, a client has to negotiate a connection to a server. Usually the client connects to the server, but this can be vice versa. Negotiating the connection defines a common language for both parties. This is better known as *a communication protocol* or simply as a *protocol*. Usually, reporting *user agent* information is a part of the communication process. The user agent information includes the name and the version number of a program.

This information defines whether the server and the client can communicate together, i.e., the client has an apt version of a program. By examining the communication information, a malicious user can gain enough information to pursue a functional attack against the service.

Evans and Larochelle [2002, pp. 8 - 9] demonstrated the use of the *splint* [Splint, 2013] tool against *wu-ftpd* [Wu-FTP, 2011] daemon [Nagpal, 2009, p. 105]. [1]. At the time their paper was published wu-ftpd was a widely used ftp daemon. In Unix, daemon is a background process that does not directly interact with a client. Splint is *a statistical code analyzer* that detects certain types of weaknesses from the program by analyzing its source code. Detecting a basic buffer overflow and misuse of functions is a part of splint's repertoire. By examining the tool's results Evans and Larochelle found new bugs, as well as bugs that were already known by the vendor. Furthermore, as the analyzed program was interacting with other computers, it was possible to use the extracted information to craft an exploit against the other daemons in the Internet.

Anybody can review open source software and review the code, use tools like splint to analyze the code and to generate an exploit. This can be seen as a shortcoming, as people can read the code and spot flaws more easily. On the other hand it is an advantage, since code gets more code reviews and bugs will likely get fixed. This certainly needs interaction with the community. Unless explicitly mentioned, all examples in this thesis are related to a Unix based operating system and C code.

3.1 Memory

Different kinds of attack methods that can be pursued via computer memory are reviewed in this section. Moreover, this helps to gain better understanding of how protection methods act.

It is easier to understand a process memory layout by having a concrete real life example. Figure 3.1 and Code fragment 3.1 show an example given by Wilander and Kamkar [2000, p. 3].

By examining the source code in Code fragment 3.1, it can be concluded that arguments, constants and variables are stored in different places in the stack (see Figure 3.2). For instance, statically declared integer variables are stored in the *Block Started by Symbol* (BSS) segment (Figure 3.1). Machine code is the

[1] A daemon process refers to a process that does not directly interact with a user. Daemons are also referred as servers.

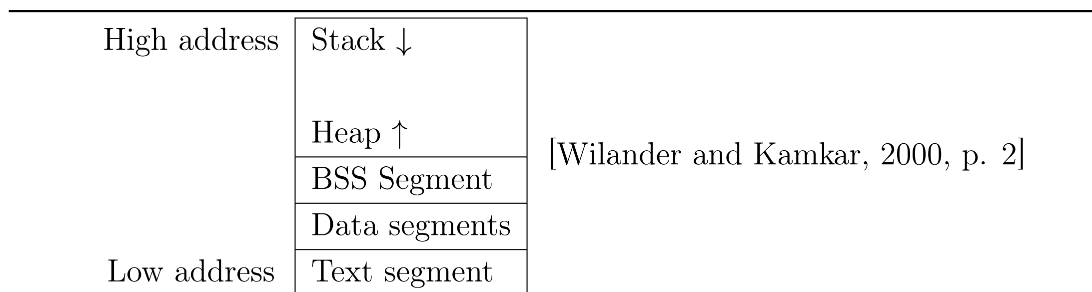


Figure 3.1 Memory layout of the Unix process.

divergent in the memory layout that cannot be observed from the source code example. Machine code is stored in the text segment.

```
static int GLOBAL_CONST = 1; // Data segment
static int global_var; // BSS segment

// argc & argv on stack, local
int
main(int argc, char *argv[])
{
    int local_dynamic_var; // Stack
    static int local_static_var; // BSS segment
    int *buf_ptr=(int *)malloc(32); // Heap
}
```

Code fragment 3.1: Memory layout elements in code.

Depending on the chosen *application binary interface* (ABI), stack grows up or grows down. Every time a function call occurs, the stack grows by one (function call will be the top element in this implementation). Each of these calls includes local variables, old base pointer, return address and arguments. In these examples, stack frame growing from high to low.

Various attack methods concentrate on altering the stack frame in such a way that the attacker can execute arbitrary code and potentially gain higher privileges. Usually, an attack is performed by altering the return address in the stack in a way that it points to a wanted memory location. In this way, malicious attacker can choose an arbitrary C standard library function to be called.

Low address	
	Local variables
	Old base pointer
	Return address
	Data segments
High address	Arguments

Figure 3.2 A stack frame.

3.2 Input validation problem

Almost all programs need some sort of an input to work. Afterwards input is handled and modified by the program. For example, in calculators input is numbers and operands. It is important to note that input does not necessarily involve keyboard activity. Sanitizing input from unintended activities is far more complicated than one could expect [Stuttard, D. and Pinto, M., 2008, pp. 19 - 20]. By altering the input an attacker may be able to set the application in a indeterministic state and perform operations that can be harmful for the system, its users or the stored data.

What makes input validation remarkably difficult is the selection of validation methods. For reading the date of birth, one can set the input to accept digits only, for example. Another input that reads digits must handle negative values correctly, and therefore it should read dashes too. A closer look into date and time parsing reveals how complicated task input validation can be. If the given input format is ISO 8601, more precisely in *YYYY-MM-DD* form [ISO 8601, 2013], inputs such as *2012-24-2* and *2012.2.24* would be illegitimate. Although the date would be syntactically right, a leap year check must be still performed. Moreover, Stuttard and Pinto note [2008, p. 20] that in addresses there can be dots, punctuation marks and other special characters that cannot be omitted, when the input is read.

What happens if the attacker is able to inject 20 characters in a field that has length limit of eight digits? There are many different routes where this could lead. If C language is assumed it can allow arbitrary code execution through a buffer overflow. Or if the attacker uses different encoding in the input and the sanitizer is not able to parse the input correctly? This could cause missing a null character and hereby many string functions would read memory until null occurs. That can and would likely be outside of the buffer and cause malfunction

of the program. More detailed explanation about buffer overflow on page Buffer overflow.

Applications use API calls for certain operations, for instance opening files with *fopen* and *open*. Many of these calls lack certain security measures and are prone for input validation attacks. Commands *execvp* and *system* are used to run binary files with an input as parameters. A foolhardy programmer could use *execvp(ls argv[1])* through a *common gateway interface* (CGI) to list directory contents. By injecting the command *; rm -rf /* in the input, it becomes genuinely harmful. If the command is run by a super user it is even more malicious. It will first list the directory contents and then execute *rm -rf /* which will remove all the files from the system. Alternatively, attacker could launch a more sophisticated attack that alters firewall rules, installs a trojan horse or joins the computer as a part of a botnet, for exapmle. Botnet is a collection of slave computers that are controlled by a malicious instance.

3.3 Buffer overflow

Over decades buffer overflow attacks have been populating the software security flaw top-lists [CWE-120, 2013]. Buffer overflow occurs when a process writes to a memory address out of an allocated area. Buffer overflow enables code injection exploitation through the current process. This is known as an arbitrary code execution. A determined attacker can gain himself or herself at least the same privileges that the process has.

Stack smashing is similar to buffer overflow but it alters stack in a way that function's real return address gets overwritten by a random return address [Ericksson J., 2003, pp. 23 - 24]. With a careful planning a malicious attacker can choose an arbitrary return address which points to a wanted function call (i.e. *system()*).

```
1 char mymsg[2] = "hi";
2 strcpy(mymsg, "hello world!");
3 printf("\%s\n", mymsg);
```

Code fragment 3.2: Simple buffer overflow example in C language. Only the relevant parts are shown.

In Code fragment 3.2, line 1 contains the first flaw: length of the *mymsg* string is 2 but strings should be terminated by the *null-terminating character* \0. null-termination is crucial, since many functions in C depend on that. Without the termination character functions do not know where to stop. Function return happens when a memory address contains the null-termination character. The character can be further in the stack, outside of the process' allocated memory area.

In line 2, the strcpy function is used for copying the string "hello world!" into *mymsg*. Since the reserved size of the *mymsg* buffer is smaller than the copied string "hello world!", buffer overflow occurs. This makes the program indeterministic.

Although one certain type of overflow was examined in Code fragment 3.2, several variations of buffer overflow exist. De Raadt [2011] lists the following variations: stack overflows, data segment overflows, *Global Offset Table* (GOT) / *Procedure Linkage Table* (PLT) overwrite, jumping to data that an attacker can execute, four byte modification possibilities and four byte read possibilities.

3.4 SQL injections

SQL is a language for relational databases. A relational database is a collection of data structures and algorithms that stores information efficiently and provides a powerful way to search and join this information together. Databases can store information of age, username, password or computer's IP address, for example. Databases are used widely in web development. [Wilton and Colby, 2005, pp. 11 - 14]

One of the commonly used attack techniques is an SQL injection attack [SQL Injection - OWASP, 2013]. Crafting SQL injection attack does not need any special skills or familiarization with the system, and is therefore easy to deploy. Nevertheless, it can be very effective since an attacker can obtain passwords, credit card numbers and other sensitive data fairly easily. Naturally with username and password combination the attacker is able to sign in into the system and gain more information from the running system and its environment.

As mentioned earlier, web development utilizes databases in large-scale. In web development form fields, for example, are often connected directly or indirectly to a database, hereby play a role of being the input for the program. These fields are parsed and passed to an SQL server which processes them later on. In a wider aspect, this is a problem that has been already investigated in the previous

section, input validation problem.

Code fragment 3.3 shows a simple SQL authentication query [Poll E., 2011c, p. 11] that is used to check whether the user has sufficient credentials to sign in into a system.

```
1 $result = mysql_query(
2             "SELECT * FROM users
3                         WHERE user_id = '$name'
4                         AND password = '$passwd';");
5 if (mysql_num_rows($result) > 0)
6             $login = true;
7 /* Parsed SQL */
8 SELECT * FROM users WHERE user_id = 'johndoe' AND password = 'password123';
```

Code fragment 3.3: A simple code snippet demonstrates an SQL authentication query.

The parsed SQL code can be seen in the Code fragment 3.3, line 8. Now, however, if one intends to act malevolently against the system the task is fairly trivial. Since *username* and *password* fields work as the input, malicious code can be injected via these fields.

```
1 /* SQL Injection through username */
2 username = ' OR 1 = 1; /*'
3 /* Parsed SQL */
4 SELECT * FROM users WHERE user_id = '' OR 1 = 1; /*' AND password = ''
5 /* Effective SQL */
6 SELECT * FROM users WHERE user_id = '' OR 1 = 1;
```

Code fragment 3.4: Infected version of the simple SQL authentication query. Dialect of SQL is MySQL.

By injecting code *' OR 1 = 1; /** in the username field malicious attacker can get access without entering a correct password. Code fragment 3.4, line 4, illustrates how system parses the query. The effective query is seen in the line 6. This is the one that the system will finally run. [Poll E., 2011c, pp. 14 - 15]

After the malicious user has gained access into the system it is possible to alter billing information, medical records or whatever information the database holds.

4 Proactive methods

This section educates what can be done generally to prevent bugs proactively. Utilizing design principles does not require support from the language, operating system or development environment, and can be therefore applied to any platform. In some principles, some of the system calls might be missing in certain environments, but usually these can be circumvented somewhat easily. This is a real advantage when compared to security frameworks that are usually designed to work in the specific environments only.

4.1 Design principles

Several design principles that are known to improve error handling, increase the safety and the controlling abilities should be used in security oriented programming. Obeying these principles does not guarantee flawless programs but helps to cope with the problem. These principles should be usually applied at the beginning of the design process, since it is easier to design functionality around them. It is viable to adopt the methods later on but usually it means significant changes into the code.

4.1.1 KISS

KISS is an acronym for Keep It Simple, Stupid! [The Free Dictionary, 2012]. The principle encourages simplicity over perfection. According to the principle, more lines of code implies more obscure structures and therefore makes it abstruse. Updating the code becomes more complicated and implementing new functionality can be challenging. In general, the more lines of code, the more bugs and (security) flaws. Moreover, program functionality becomes less deterministic and ambiguous at many levels. The original Unix was designed to adhere to the KISS principle, and therefore one tool is designed to accomplish one operation.

4.1.2 Whitelists

Whitelist is the opposite of blacklist. Instead of denying forbidden functionality whitelist allows certain functionality or operation. If a user or a program does not meet the requirements, it is denied by default.

An excellent usage of whitelisting is presented in the program *TCP wrappers* [TCP Wrappers, 2011]. TCP wrapper is used for limiting hosts to access services.

As mentioned in the *input validation problem* section, one should never trust any user input [Stuttard, D. and Pinto, M., 2008, p. 19].

4.1.3 Least privilege

When running a process, it should not use higher privileges than is needed. If possible, process should drop all extra privileges. For example, HTTP daemon should not run as root once the socket port binding and possible chroot has been completed successfully.

Example 4.1.3.1 HTTP daemon needs to bind to port 80 (HTTP traffic) in order to be able to serve web pages for web browsers. Ports that are equal to or lower than 1023 are called *low ports*. In order to bind a socket to these ports, *super user* privilege [1] is required to accomplish the task. After completion the port binding daemon should drop all extra privileges that are no longer needed.

However, as Provos et al. [2003, p. 2] state, this does not guarantee safeness. Certain type of flaws in a daemon process can still leak higher privileges to a malicious attacker. As a countermeasure, Provos et al. propose *privilege separation* as an addition to the least privilege. More details about privilege separation can be found below.

4.1.4 Fail securely

When a malfunction happens it must be taken care in an appropriate manner. If one level of security fails there should be another level to mitigate malfunctioning (*defence in depth*). For instance, a computer that is linked to the Internet should not solely rely on a firewall [Poll E., 2011a, p. 9]. It is even more crucial when the computer runs services.

Code fragment 4.1 leaks several pieces of important information to an attacker. By knowing the database and table names it is possible to try database related attacks like SQL-injection to gain access into the system. The path name reveals the running operating system, which in this case is likely Windows. Here call stack trace (see Appendix A) reveals the used programs and platforms, MySQL and ColdFusion. The previous information might be enough for the malicious attacker to penetrate into the system or crash the site.

The previous case happened to the author in a particular site. After reporting misconfiguration to the site administrator it had no effect nor the administrator

[1] *Root* user has the highest privilege. Also, ID number 0 refers to root.

replied to an email. This is a good example how security is seen as a low priority task by some administrators. An appropriate way to handle debug and system messages is to write errors to a log file instead of showing them to users [Poll E., 2011a, pp. 35 - 41].

In this paragraph, services refer to daemon processes. These processes could be web servers, dns servers or proxy server, to name a few. Previously mentioned processes are usually responsible for important functionality such as running e-commerce or relaying traffic between networks. Since these daemons should be always running it is crucial to minimize their downtime. There are several monitoring software to perform certain actions when the daemon process goes down. This type of monitoring software is called *process supervision*. *Monit* [2013] and *Supervisor* [2013] represent such daemons. Although the idea is somewhat good, there are certain flaws. What happens if the server malfunctions and process supervision daemon starts it again continuously? Or it ends up forking new zombie processes and finally the computer is rendered unusable because of resource exhaustion? Although these programs can be used to handle malfunctions, there is a high probability that they cause malfunctions rather than provide fixes. The correct approach would be to fix the root cause instead of masquerading it.

4.2 Compiler and language based safety methods

Machine language is numerical code, which computer's *central processing unit* (CPU) executes directly. It is hard to read and understand for humans, since it neither obeys the structure of a human language, nor has similar lexical or syntactical expressions. *A symbolic machine language* remedies this shortage. For example, by using a certain symbolic machine language multiplication operating is expressed as MUL [Tremplay J-P and Sorenson, 2008, pp. 1 - 2]. Syntactically it is closer to *natural language*, though it differs a lot lexically. Computer languages can be split into two different categories, *low level languages* and *high level languages*.

Both machine and symbolic machine language are a part of the low level language category. These languages have strict grammars. Vice versa, languages that human use for communication are more subtle, ambiguous, loosely defined and might vary a lot syntactically. As low level languages have a simple instruction set, it is hard to accomplish highly abstract tasks like implementing a B-Tree data structure. For this purpose there are higher level languages that are more subtle as compared to their predecessors, low level languages.

```
You have an error in your SQL syntax; check the manual that corresponds to
your MySQL server version for the right syntax to use near '

and afdeling.provincie in (0,3) order by afdeling.sort' at line 3

The error occurred in D:\websites\kicker\content\Uitslagen\index.cfm: line 32

30 : select distinct(afdeling.afdelingid) as afdelingid, afdeling.afdeling
     as afdeling
31 : from reeks left join afdeling on (reeks.afdeling =
     afdeling.afdelingid)
32 : where seizoen = #qGetseizoen.maxseizoen# and afdeling.provincie in
     (0,#regio#) order by afdeling.sort;
33 : </cfquery>
34 : <cfoutput>

SQLSTATE 42000
SQL select distinct(afdeling.afdelingid) as afdelingid,
             afdeling.afdeling as afdeling from reeks left join afdeling on
             (reeks.afdeling = afdeling.afdelingid) where seizoen = and
             afdeling.provincie in (0,3) order by afdeling.sort;

VENDORERRORCODE 1064
DATASOURCE kickersql
```

Code fragment 4.1: An example of an inappropriate way to handle errors.

In a design process, one should carefully choose the implementation language. The language that guarantees *type safeness* [Poll E., 2011b, p. 20], *integrated sandboxing* and is pointer free, should be always preferred. By using a language that implements type safeness it is impossible to mix strings and integers on the following way: $2 + "\#"$.

Misuse possibility exists if the language offers memory operations via pointers [Poll E., 2011b, p. 21]. Therefore pointer free (*memory safe*) languages should be preferred. Occasionally a program has to be implemented in a language that does not fulfill these security needs. In these cases, functions that are boundary-aware should be preferred and function return values should be verified in the case of errors. Also, function return values should always be checked against errors if the language does not implement exception handling itself.

In low level languages, an attacker can craft an exploit that is highly dependent on machine architecture instructions and bypass higher level language limits and safety methods. Albeit this is an interesting topic, the main focus of this study is in the higher level languages.

Majority of the open source operating systems are implemented in C language because it has support for a symbolic low level language. It is possible to write hardware drivers and make certain functions faster with lower level implementations. C language's main strength is its speed and simplicity. The language has many functions that gain speed by omitting boundary checks and by accessing memory directly via pointers. Omitting boundary checks and offering bad APIs can be seen as a curse of C security-wise, though fortunately there are safe and clean versions available. Although the speed assumption is true in general cases, Miller and de Raadt [1996, p. 2] implemented safe functions without a critical speed regression.

One of the functions that lacks boundary checking is *strcat*. This function is used for string concatenation. Boundary-aware, portable and widely adapted version of strcat exist, *strncat*. A manual page for strncat regarding the Open Group Base Specification strncat is seen in Code fragment 4.2. Prototype of strncat takes exactly three arguments: $s1$ (destination), $s2$ (source) and n (size). If a source buffer size is greater than or equal to a destination buffer, null-terminating string is omitted. To use strncat safely, Miller and de Raadt [1996, pp. 1-2] encourage to always copy size of $n-1$ and null-terminate string by hand, although in some rare cases the previous operation is exaggerated.

Miller and de Raadt also disclose the misuse of strcnat. Particularly the size parameter is often thought to be the size of the destination buffer. Miller and de Raadt use the following formalization to clear this misconception:

> *Most importantly, this is not the size of the destination string itself, rather it is the amount of space available.*

As their last concern [1996, p. 3] pointed out that the performance regression made by the boundary check is minor. During the time the paper was published (1996), computers had significantly lower performance. Today when CPUs have more than tripled their computing performance and can do more than one operation in a single instruction, this claim is considered outdated.

```
SYNOPSIS
    #include <string.h>

    char *strncat(char *restrict s1, const char *restrict s2, size_t n);

DESCRIPTION
    The strncat() function shall append not more than n bytes (a null byte and
    bytes that follow it are not appended) from the array pointed to by s2 to
    the end of the string pointed to by s1. The initial byte of s2 overwrites
    the null byte at the end of s1. A terminating null byte is always appended
    to the result. If copying takes place between objects that overlap, the
    behavior is undefined.
```

Code fragment 4.2: Unix manual page of strncat (only the relevant parts are included).

4.2.1 Automatic memory management

There are two types of variables: local and global variables. A local variable has visibility in a function block. A global variable can be accessed anywhere from the same file. Many languages have adopted a syntax where *scope* is defined as an area between the curly brackets, { }. Code fragment 4.3 depicts the previous situation.

```
1  variable A // Global variable
2
3  function()
4  {
5          variable B // Local variable
6  }
7
8  main()
9  {
10         print(A) // Since A is global, this works
11         print(B) // Does not work, since B is local under function()
12 }
```

Code fragment 4.3: Local and global variables.

When a program returns from a function any of the locally reserved variables cannot be accessed anymore (excluding global variables). The lifetime of a

variable is consequently as long as the program execution stays in the scope.

Exceptions exist depending on the chosen programming language. For example in C language, allocated resources are retained after function call returns – if resources are not explicitly freed. Apart from the previous exception, all dynamically allocated memory should be freed before leaving the function.

However, the previous approach has certain drawbacks. If one forgets to release the dynamically allocated memory before returning from the function, memory stays allocated and reference to it is lost. In case the allocated information is confidential, there is a possibility to leak this information. A determined attacker is able to read unwanted parts of the memory, and therefore one should pay attention how memory is aligned and zeroed when confidential information is stored.

In certain languages that do not provide automatic memory handling, memory can be allocated without first zeroing the area. C is one of these languages and to be specific, its function *malloc* can be used for the purpose. With certain knowledge the attacker can alter the previously used memory block and read confidential information. One possible defence against these kind of attacks is to patch malloc to force erase allocated memory blocks. Such functionality is provided via *malloc.conf* (J and Z switches) at least in FreeBSD [FreeBSD malloc, 2014] and OpenBSD [OpenBSD malloc, 2013]. An other approach would be to switch to a memory safe language.

Automatic garbage collection takes care that all dynamically loaded chunks are freed afterwards and the previously mentioned leak cannot happen. Without automatic garbage collection every dynamically allocated memory block must be released by the user. Omitting the deallocation operation makes it possible to have a memory leak.

4.3 Protecting the memory

There are several ways to protect memory from leaks and buffer overflows. In this section, the main focus is the *ProPolice* memory protection method. It prevents buffer overflow and stack smashing attacs moderately well as proven by Wilander and Kamkar [2000, pp. 13 - 14]. These will be discussed in greater detail under Predefined flaws section.

Figure 4.1 shows an enhanced process memory layout that resides in a computer memory. The technique is known as ProPolice [IBM Research, 2001] and it protects against the *stack smashing attacks*, also known as stack overflow attacks.

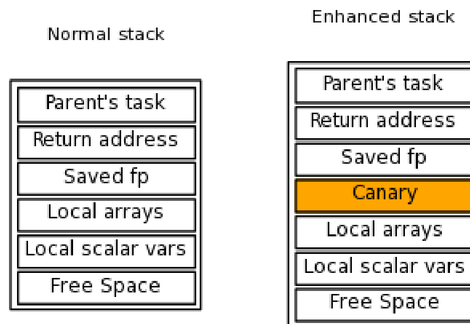


Figure 4.1 Structure of a computer memory stack and an enhanced memory stack.

Protection works by inserting an extra information, canary (sometimes called as a cookie), into the stack. Canary is basically a random value. The canary is inserted into each function call at compile time [Advances in OpenBSD, 2013, pp. 10-12]. If the canary changes during the execution time, the program terminates.

Furthermore, ProPolice reorders the stack in a way that the flags and the pointers are placed lower in the stack. If overflow occurs, it likely first overwrites the canary. It makes harder to overwrite flags and pointers since the canary changes are noticed. ProPolice has moderately low regression to performance since it is only about 1.3%.

Even if the compiler would use canaries to protect against stack smashing, there are still possibilities to pursue an attack. To make buffer overflow attacks even harder to gain any benefit, the OpenBSD [OpenBSD, 2013] operating system [Advances in OpenBSD, 2013, p. 16] introduces randomizing *ld.so* (run-time link-editor) memory load locations, as well as randomizing *mmap* and malloc calls.

Another widely used approach against buffer overflow attacks is to use an *NX-bit*. NX-bit stands for Non-Execute Bit. It works by denying *write right (w)* and *execute right (x)* existing for a memory stack at the same time. Its importance have been seen so crucial that modern processor manufacturers has implemented it on a hardware level. Although it is widely used and works in general cases, it can be circumvented. A proof of concept attack was introduced by Mastropaolo [Mastropaolo M., 2005].

4.4 Access control methods

Many different access control methods exist and it can be tedious to find an appropriate application for each method. It should be clear for what purpose access control is going to be used. Some environments work perfectly with *discretionary access control* (DAC), whereas other environments demand more fine-grained approach and control of information flow. Discretionary access control is a basic access control and is shipped with most operating systems. Object can have only one user and one group set at time. It is also easy to implement for different systems. If the information flow is important, then *role based access control* (RBAC) methods like *mandatory access control* (MAC) should be considered.

Some of the methods are too complicated to be used consistently. *Access Control List* (ACL) is a good example of ambiguous functionality where capability setting is not straightforward. For instance, file or directory can have read and write access for user John, execute and read access access for user Jess and read for the other users. Files that have many distinct rules will become quickly unmaintainable.

Another example is from the Windows and the Macintosh OS X world where MAC restrictions (on by default) displease many regular users by not letting them complete their tasks. After downloading a file from the Internet a dialog asks whether the program should be executed. More similar pop ups will follow and the screen is filled up with questions − just to complete a simple task. Piece by piece users turn off the MAC based access control and are again vulnerable to exploits.

4.4.1 Sandboxes and chroot

Sandboxing is an important factor when isolating running services or processes. Well designed sandbox implementation mitigates compromising the complete system in the case of a malfunction of a program.

The chroot (*change root*) function call changes the visibility of a given directory. If a directory $/home/user1$ is influenced by chroot, it is seen as /, the root directory. Chroot is implemented in all modern operating systems and does not need any extra framework to be installed in order to gain a simple and working file system sandbox.

There are few drawbacks when chrooting a process. All the needed runtime dependencies must be copied inside the chrooted directory. Since $/home/user1$ is now seen as /, the file system hierarchy must be imitated inside the chrooted

environment. References to an object that resides outside the chroot do not work. For instance, soft links pointing out of the chroot are superfluous. If the process must run with root privileges, chroot is rendered useless [Provos et al, 2010], since root can always escape from the chroot environment.

Example 4.4.1.1 Keeping a compilable program updated in a chrooted environment can be tiresome. If the operating system updates its *libc*, one might have to update libraries in the chrooted environment too, or otherwise programs can malfunction. The program *ldd* (list dynamic object dependencies) is a helpful tool when replicating programs into the chroot environment. Copying the needed libraries can be eased with scripting which omits manual intervention.

4.4.2 Privilege separation

In a basic non-secure environment, clients interact directly with a privileged process. This exposes the system to a danger, since a malfunction can leak higher privileges or grant access to confidential information. Since it is not always possible to drop higher privileges, Provos et al. [2003] implemented privilege separation. They propose isolation between the interacting clients and the server processes. By this way clients interact with non-privileged processes that have been forked from the privileged parent process. The privileged parent process can be needed for crucial tasks, like binding the listening address to low ports. Low ports are smaller than 1024. Splitting the processes makes the program more resilient against attacks since a malfunction only affects the forked children. An outline in Figure 4.2, shows how communication is made when a client from the Internet requests an HTTP page.

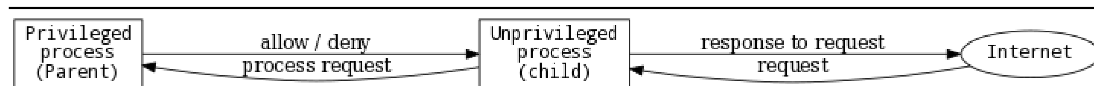


Figure 4.2 Communication negotiation between an Internet client and a service where privileges have been separated.

5 Static code analysis

Expressing errors clearly is becoming more important since the complexity of programs is growing. Bug hunting in such systems is tedious and difficult. Automated checkers and analyzers exist for different purposes, for example scanning vulnerabilities or measuring the complexity of the code. Static code analyzer is a compiler-kind program that performs lexical analysis of source code and computes a variety of measures. Hereby, bugs and vulnerabilities can be exposed without running a program binary [Cauldwell P., 2008, p. 135].

Since analyzing code by hand is a vastly time consuming and error prone task, static analyzers can outperform humans. However, that does not remove the human factor from the analyzing loop since not all the issues mean real errors or vulnerabilities. These false alarms are called *false positives.* Besides that, there are many cases where humans are able to find errors when analyzers detect nothing aberrant. Patrick Cauldwell [Cauldwell P., 2008, p. 135] encapsulates usage of static code analysis as follows:

> *That knowledge comes at a cost, however. It is easy to gather more information than you know what to do with using such tools. To make the best use of static analysis tools takes time and study – time to get to know how to use the tools properly.*
>
> *Results can easily be misinterpreted if you aren't familiar enough with what they mean. There is also a danger of "coding to the numbers," or changing your code to make metrics look more favorable without adding any real value.*

5.1 Types of checks

To give an idea of what kind of flaws and vulnerabilities a bare static analysis is able to detect, one can have a look at *Coverity's* free online scanner issue list [Coverity, 2013]:

- **Resources leaks**
  File or socket handles that are not freed after use. [Resource Leak - OWASP, 2013]
- **Dereferences of null pointers**
  Trying to use a memory location through a null pointer. [Null Deference - OWASP, 2013]

- **Incorrect usage of APIs**
  One could use *chroot* without calling *chdir* afterwards. Syntactically this is right, but *chdir* call must be made, since otherwise an attacker can escape from the restricted environment. Mitre [CWE-227, 2013] explains that in the following way "An API is a contract between a caller and a callee. The most common forms of API misuse occurs when the caller does not honor its end of this contract."

- **Use of uninitialized data**
  C language does not initialize its stack variables by default and hereby uninitialized variables mostly contain garbage with the stack memory references. It is possible an attacker may alter this data. [CWE-457, 2013]

- **Control flow issues**
  A program does something that was not intended by the programmer, such as a logic error. Mitre [CWE-670, 2013] mentions an example where a multi line if condition is not enclosed in brackets.

- **Error handling issues**
  There are several improper ways to handle errors. The worst of all is to omit error handling completely. In C, this means omitting checking function return values and an external variable (*errno*) which is set by system calls. [CWE-388, 2013] [CWE-391, 2013]

- **Incorrect expressions**
  A programming clause is syntactically correct but logically incorrect. Mitre has a list [CWE-569, 2013] of cases that are related to incorrect expressions. One of the shown incorrect expressions is ID 481: *Assigning instead of Comparing*. A simple example of erroneous comparison: $if(a = b)\{dosomething\}$. Another equality mark is needed to perform the correct comparison.

- **Concurrency issues**
  A common concurrency problem is *time of check, time of use (TOCTOU)*. Since multicore hardware platforms are more common nowadays, it is reasonable to utilize all cores and processors simultaneously. This means that many problems can be divided into smaller parts and run in parallel, *threaded*. Threads can communicate together by sharing memory but there are also certain drawbacks in this method. When multiple threads access the same resource without correct locking or implementing atomic-

ity operations [CWE-622, 2013] a race condition may happen. Afterwards when another thread reads the value, it may be something that was not indented by the programmer.

- **Insecure data handling**
  Better known as input validation problem. When the user input data is not handled properly it might trigger unwanted commands. In demonstrative examples, Mitre mentions [CWE-20, 2013] that if prices are not checked against negative values, a user can alter the total sum.

- **Unsafe use of signed values**
  C has several variable types. There are, for example, *signed int* and *unsigned int*. Since unsigned cannot be negative, there can be logical errors when operating with two different types of signedness. [CWE-570, 2013]

Although the list is rather short it only covers some cases that Coverity scan is able to detect. Plain C related vulnerabilities list [CWE-658, 2014] is quite comprehensive. The list does not include vulnerabilities which are common for all languages or platforms, because the list would be enormous. Checking the list manually would be cumbersome and error prone assignment.

It is worth to note that *dynamic analysis* [Hass A. M. J., 2008, p. 381] is contrary of static analysis. The main idea of these tools is the same: detect misuse, although the functionality differs. The biggest difference is that programs are analyzed during execution time. These tools detect anomalies in memory usage by analyzing allocation and deallocation events. *Valgrind* [Valgrind, 2013] is a such tool that is able to detect these kind of bugs but also thread related problems and race conditions. Advanced memory inspection includes pedantic analysis of pointers and memory, and it can predict memory exhaustion before it occurs.

5.2 Annotations

An analyzer that supports for annotations, has finer finer granularity to track content and perform more precise checks. In some analyzers, annotations are enclosed within the comments whereas other analyzers use macros. Annotations are added into the source code and the analyzer is able to provide stricter control flow checks against the given variable or object. It is possible to indicate that a certain parameter in a function is read-only, therefore writing into the parameter

cannot be done in this function. Code fragment 5.1 has an example showing how annotations can be used.

```
1 int
2 myfunc(_IN_(mystring) char *mystring, size_t len)
3 {
4         mystring = "test"; /* Causes failure */
5 }
```

Code fragment 5.1: Code with annotations.

Syntax of the annotations will vary regarding the used analyzer. Here in Code fragment 5.1, *_IN_* means that the function can read from the buffer *mystring* but writing will cause an issue for the analyzer.

5.3 Control flow and data flow analysis

Some of the analyzers perform *control flow analysis* and *data flow analysis.* Data flow analysis tracks down variable usage at different points in a program execution [Cooper K. D. et al., 2013].

Code fragment 5.2 demonstrates how control flow analysis works in practice. In general, control flow analysis has only two outcomes, true or false. When function call returns before all dynamically allocated resources are freed, control flow analysis results false. If there is an possibility for alternative *return* before calling *free*, the result is false and a possible memory leak exist. It depends on the analyzer's implementation how well it catches the leaks. Some analyzers use more fine-grained heuristics which produces better results, namely reduces false positives.

Demonstration of data flow analysis is depicted in Code fragment 5.3 [Poll E., 2011d].

5.4 Survey of analyzers

This section familiarizes with different tools that are freely available for everyday use. Some of the commercial brands offer analyzers free for individual use. Anne Hass [Hass A. M. J., 2008, p. 223] mentions in her book that *some standard development tools, such as compilers or linkers, are able to perform limited*

```
1  void
2  dosomething(int multiplier)
3  {
4          char *myarr = NULL;
5
6          if ((myarr = calloc(BUFSIZE, sizeof(char))) == NULL)
7                  err(1, NULL);
8          if (multiplier < 10)
9                  return; /* returning without freeing myarr */
10         /* ... do something in here ... */
11         if (myarr)
12                 free(myarr);
13 }
```

Code fragment 5.2: An example of control flow analysis depicting a possible memory leak. Variable myarr is not freed in some circumstances.

```
1  /* Case 1 */
2  int d = 5;
3  c = d; /* Variable c is initialized */
4
5  /* Case 2 */
6  int c;
7  c = d;
8  d = 5; /* Variable c is not initialized, thus containing random data from the memory */
```

Code fragment 5.3: Data flow analysis.

static analysis. The book was written in 2008 and nowadays many compilers and compiler suites have more advanced static code analyzers than in 2008.

An analyzer can be driven moderately easily by utilizing *Makefile*, for example. If there is no Makefile support it can be an exhausting process trying to keep includes, libraries and other related functionality in order. If the analyzer needs tremendous work to operate and to keep it up to date with the changing code base it will become harder to integrate it into the development cycle. Cumbersome setup will likely kill the motivation to use the analyzer.

When analyzing a result, too many false positives will hide the real problems and the results are rendered unreadable and non-functional. Consequently it is crucial to have an easily integrable analyzer that gives helpful notifications of how to reduce the bug count. Pointing a possible bug is certainly good but giving a

hint how to fix it is even better.

The used tools are shown in Figure 5.1. More detailed description of each analyzer and the options used in the tests.

Analyzer	**Version**	**Annotations**	**License**
Clang, LLVM	3.5	Yes	University of Illinois/ NCSA Open Source License
cppcheck	1.67	No	GPLv3
GCC	4.8.2	Yes	GPL3+
Flawfinder	1.27	No	GPLv2
Splint	3.1.2	Yes	GPL

Figure 5.1 List of tested static code analyzers.

5.4.1 Clang and Scan-build

Clang [Clang Analyzer, 2013] is a C language front end of the LLVM compiler suite [LLVM, 2013] which is sometimes called *Low Level Virtual Machine*. The name is misleading since it has little to do with traditional virtual machines. LLVM is *a collection of modular and reusable compiler and toolchain technologies*. Due the modular nature of Clang it is possible to implement a decent static code analyzer for it.

Clang ships with a separate static analyzer called *scan-build* [Clang Analyzer, 2013]. Several separate analyzer modules exists for different uses. Most of the modules solve a single issue, such as checking stack overflows or following malloc() and free() usages. Both LLVM and scan-build have support for annotations. Output of scan-build is HTML.

5.4.2 Cppcheck

Although the name cppcheck [cppcheck, 2013] refers to the *C++* language, it is capable to parse the C language. Cppcheck has support for the following features: Out of bounds checking, check the code for each class, checking exception safety, memory leaks checking, warn if obsolete functions are used, check for invalid usage of *Standard Template Library* (STL), check for uninitialized variables and unused functions. False positives can be suppressed with comment annotations.

Cppcheck also includes additional checks against style, performance and portability. Portability concerns checking against certain platform related conventions.

There is an option for XML output which makes integration to other tools, such as continuous integration systems easier. During the development process same piece of code is often edited by different developers. At times when a user A and a user B combine their code changes together a merge conflict occurs. When the conflict occurs it must be resolved by the code committer. Continuous integration automates testing on these situations by ensuring the code works as it should. HTML output is available via plugins.

5.4.3 Flawfinder

Flawfinder [Flawfinder, 2013] is a somewhat crude analyzing tool written in Python to clean up *at least some potential security problems.* It tries to be a small as well as easy to install and use analyzer.

Flawfinder has a database of security wise poorly designed functions. Its documentation [Flawfinder, 2013, How Does Flawfinder Work?] mentions the following functions as an example: Buffer overflow: *strcpy, strcat, gets, sprintf,* and the *scanf* family. Format string problems: *[v][f]printf, [v]snprintf* and *syslog.* Race conditions: *access, chown, chgrp, chmod, tmpfile, tmpnam, tempnam,* and *mktemp.* Potential shell metacharacter dangers: *exec family, system, popen),* and poor random number acquisition (such as *random*).

Later on the source code is matched against these flaky functions and sorted against the severity of vulnerability. Then the riskiest flaws are set on the top of the results. The risk level is shown in square brackets from 0 to 5, where 0 stands for *very small risk* and 5 stands for *great risk.* Besides functions, their parameters are considered in the ranking. A function with constant parameters does not have that high impact for the ranking as mutable parameters, for instance. Flawfinder is not aware of function parameter data types and cannot determine whether the function is called correctly. There is no support for data flow analysis. By default output is textual but HTML output can be enabled with a run time switch.

5.4.4 GCC

GCC stands for *Gnu Compiler Collection* [GCC, 2013]. For a long time GCC has been the de facto compiler in the open source compiler scene. Nowadays GCC has gotten a worthy competitor from Clang. GCC has support for variety of languages and hardware platforms. GCC was chosen for the comparison as a control analyzer. Although it supports checks that are common to static analyzers the main functionality lies on compiling and linking.

An article in LWN.net [Corbet J., 2012], explains the porting efforts of static analyzer to GCC. Earlier versions of GCC had more vague error messages and they were not as readable as depicted in the Figure 5.4. Albeit the warning is somewhat easy to spot in here it can be indicated better as will be seen soon. Modern versions have this ambiguous behaviour remedied and somewhat copied the outcome from Clang. LWN's article mentions

> *Some developers at Google have been working on just such a project for some time, but they have just relocated the project from GCC to the LLVM Clang compiler, saying that GCC is not suited to the work they want to do. The result has been a sort of wake-up call for GCC developers.*

One interpretation could be a risen need of a better static analyzer, as a part of the compiler tool chain. Now, even the compiler tool chains must do some basic checks that are comparable to analyzers. Modular compiler makes it possible to write more advanced analyzers.

```
1 unsigned long a = 10;
2 signed long b = −10;
3
4 if (a > b) /* line 25 */
5
6 /* GCC output below */
7 example_flaws.c:25: warning: comparison between signed and unsigned
```

Code fragment 5.4: GCC 4.2.1 error output.

5.4.5 Splint

Splint [Evans D. and Larochelle D., 2002] is a successor of the *lint* tool. Splint does many of the traditional lint checks [Splint manual, 2013] including unused declarations, type inconsistencies, use before definition, unreachable code, ignored return values, execution paths with no return, likely infinite loops, and fall through cases. More powerful checks are possible by using source code annotations. In splint, annotations are stylized comments that document assumptions about functions, variables, parameters and types.

Splint performs checks againts the following issues: dereferencing a possibly null pointer, using possibly undefined storage or returning storage that is not properly defined, type mismatches with greater precision and flexibility than provided by C compilers, violations of information hiding, memory management errors including uses of dangling references and memory leaks, dangerous aliasing, modifications and global variable uses that are inconsistent with specified interfaces, problematic control flow such as likely infinite loops, fall through cases or incomplete switches, suspicious statements, buffer overflow vulnerabilities, dangerous macro implementations or invocations and violations of customized naming conventions.

6 Test environment setup

This section describes the test setup and processes that were used to complete the tests. First I will concentrate on operating system specific adjustments and then familiarize with compilers related setups. Information regarding the static analyzer setups will follow. In the last section I define a variety of predefined flaws and propose fixes for them.

There is also a review for each tool regarding the difficulty of setting it up for use. For the complete list of used switches please refer to Appendix C.

The analyzers are reviewed by their usability, false positive ratio, given mitigation technique in case a bug is found and how well it integrates in the development cycle.

Probably the trickiest part was to set switches for the compilers. Simple search over GCC manual page revealed over 200 switches. For the other tools the setups were more or less straightforward.

Ranking of the analyzers would be cumbersome as it would need real world application tests and to get most of the tools, adding annotations in code. Also, each of the found issues should be reviewed with care. With a project of several thousands of lines of code it takes more time and more work to analyze results thoroughly. More pondering regarding this is in the conclusion and future studies section.

Keeping these facts in mind, it is important to remember that this review is only cursory.

6.1 Operating system

Primary tests were performed under an Openbsd operating system running amd64 [1] platform and with a current release, here 5.5-beta. Secondary tests performed under Linux Mint 16 Mate running 64 bit platform. Makefile needs adjustments if running on Linux. This is covered in Makefile. With small tweaks the tests should be repeatable on any Unix operating system.

Some of the vulnerabilities depend on operating system internals such as applied security additions and compiler switches. Buffer overflow mitigation can be integrated to the compiler and consequently entire userland can gain the advantage of this. Userland refers to programs that run outside of the operating

[1] Generally amd64 is an alias for 64 bit hardware platform.

system's kernel memory space. One of these methods is ProPolice, which was discussed earlier.

6.2 Compilers setup

Compilers contain a colossal amount of code and a bare Unix manual of GCC contains text worth of a small book. Understanding all the runtime switches and their combined effects in a larger context is almost impossible. Using different compiler switches generates varying machine language, and hereby the resulting output code can be hard to predict. This makes debugging harder and may lead to cases where code works deterministically only on certain hardware architectures. When compilers term is used in this section, it refers to Clang and GCC.

Compilers have built-in optimizers that can be enabled with switches. Automatic optimization tries to tune machine code to work more efficiently but in some situations the end result is reduced efficiency. It is possible that compiler will generate incorrect optimizations. For instance, some of the memory structures might get misaligned and this causes performance degeneration. It is also possible that optimizations can alter functionality in a way programmer did not intend to. GCC's Bugzilla (bug tracking system) has an example [GCC Bugzilla, 2007] where optimization removes a null pointer check when *-O2* optimization is used. Spotting the optimization errors demands verbatim reading of assembly code and in big projects this is not a realistic requirement.

Benefit gained by micro optimizations is often moderate. Combining all this information together with varying hardware platforms, userlands, kernel APIs and other variables, makes it significantly harder to have deterministic code and functionality.

Some of the chosen switches are not directly related to security scanning, for instance *-Wsystem-headers* shows constructs and messages that are found from the system headers and are overlapping in user's code. Afterwards this is a great aid when debugging the program since it produces less obfuscated output. All automatic optimization related switches are omitted, also omitting style related options such as *-Wparentheses*. It is worth to note, and also mentioned in the GCC's manual page that not all static analysis related switches give value for hardening the programs.

There were some serious problems setting up the Clang's scan-build. Errors were vague and searching up the Internet did not come up with a clear answer. The error was related to the *CC* variable in Makefile. If *CC* is set to something

else than *gcc* the end result is vague error and scan-build does not provide any output. This is nowadays covered by scan-build's website [Clang Scanbuild, 2014] under the section *Gotcha: using the right compiler*.

Switches that are typical to certain dialects of C are also omitted. C89 & C90 alias ansi and C99, are examples of C dialects. The chosen switches follow C99 dialect.

6.2.1 Clang's scan-build

List of available checks can be viewed with the *scan-build –help* command. By default a decent list of checks were enabled, but nevertheless all C related extra checks were included. ArrayBound checking has two versions and the newer one was used. Certain checks had alpha prefixes in their names that could be a hint of their development state. The documentation did not reveal if this assumption is right. Not all the enabled test cases are covered by the example_flaws.c but were enabled for genericity.

Overall the setup was easy, although obscurity occurs when a variable *CC* is defined in a Makefile. The result was zero flaws in the analysing report. It was time-consuming process to track down the culprit since it did not clearly state any error. It is easy to assume analyzer did not find any flaws and that analyzed source code is sound. Such errors and therefore false assumptions of sound analyzing results, are hazardous.

6.2.2 Flawfinder

In Flawfinder, there is no possibility to control which checks are enabled. The only way to affect results is to adjust the level of ranking, more precisely what is the minimum trigger level. The rest of the switches are either controlling the output or easing an IDE or editor integration. While testing the correct setup, output becomes easily unreadable by being too verbose. As a workaround, $-S$ switch (single line comments) can be used.

In general, there are not that many switches to affect the end result. This makes setup simple and swift. On the other hand extra functionality would be good to have since the analyzing is a bit harsh and terse now.

6.2.3 Cppcheck

Cppcheck setup is fairly simple. Its Unix manual page and quick start guide were clear and simple. It needed a path for the system headers, choosing the C dialect

and turning on a verbose mode. Extra checks seemed to find errors from system headers so those were disabled as it garbled the results. In overall, the setup is straightforward and fast since it ships with sane defaults.

6.2.4 Splint

Splint has a clean and simple Unix manual page that is well constructed. The manual page also gives a hint that more info can be retrieved by running *splint -help* in the command line. Running splint -help modes reveals that not all the switches are documented in the manual page and there are many switches that are not enabled by default. This mode includes a comparison chart that explains clearly which tests are included in the each mode. After few test runs standard mode was chosen. This is the default mode if no modes are selected. Any stricter mode renders results practically superfluous.

The initial setup can be done quickly and manual page is helpful source when choosing the right switches.

6.3 Predefined flaws

By using predefined vulnerabilities, it is known where the vulnerabilities lie and how those should be handled appropriately. It can also reveal if the operating system has taken extra steps to proactively plug certain leaks, i.e. using mitigation techniques against buffer overflow vulnerabilities. This gives us a controlled environment where statical analyzers are easier to test. By obeying this convention it should be moderately easy to spot which flaws or even vulnerabilities the analyzer omits.

Rationale of the chosen tests were their ranking in the Mitre's TOP 25 list [CWE Top 25, 2011] and hand picking a few personally seen flaws from the *Weaknesses in Software Written in C* [CWE-658, 2014] list.

To control false positives, there are also proper implementations of the flawed functions. The fixed functions have _fixed suffix in their name. Not all the examples are vulnerabilities by default but when combined with recursion, data structures and other complicated structures a competent cracker can gain more power than originally intended.

Test functions and their explanations are given below. In each item, the last section proposes a correct fix or fixes for the problem.

- **example_signedness(void)**
  Comparing against signed and unsigned values. This might cause logical errors since signed numbers can be negative but unsigned cannot.

  To fix the problem, change both to *signed* or *unsigned* or at least use an explicit cast when comparing to let compiler aware that the issues is noted and intended. When doing explicit casts be sure the numbers will not overflow. Also, take into account that unsigned numbers will not store negative numbers.

- **example_integeroverflow1(void)**
  Causes an integer overflow which means that a given number cannot fit in the given space. Here a hexadecimal number 0xbadbadbad (in decimal 50159344557) cannot fit into an integer which is usually 32 bits, although the size can vary as it depends on the used hardware, compiler and operating system implementation. This becomes more fatal if a user input is stored to the variable and appropriate checks are not used. According to the C99 standard integer overflow causes undefined behaviour [C99 standard, 2007, p. 4].

  Changing *unsigned int* to *long* remedies the problem. The better fix would be to use *u_int64_t* or equivalent type definition that guarantees the variable to be unsigned 64 bit number even if the hardware platform changes.

- **example_integeroverflow2(void)**
  Uses the function *atoi* to convert characters to a number. The error in here is that atoi does not perform any additional checks, which makes number overflow viable.

  With a careful use of *strtol* and *strtoul* functions one could prevent overflows. These functions use the *ernno* global variable to indicate errors. For more detailed info of occurred error can be viewed by using the *strerr* function and errno as parameter. A better solution would be to use the *strtonum* function that reliably converts chars to (long long) integer.

- **example_compare(void)**
  Compare whether float numbers are equal. Computer does not handle float numbers as humans, hereby computer does not see if 0.001 is equivalent to 0.001. Representing a real number is always an approximation, and therefore, representation of 0.001 can be 0.0009999, for example.

Comparison must be done by using certain error level where the numbers can be seen equal. Let us call the accepted error level as delta δ, and the tolerance 0.0001, hereby δ = *1.0E-4*. One possibility to fix the problem could be *if (fabs(a - b) < 1.0E-4)*. Java for example, uses the similar approach that was presented here.

- **example_bufferoverflow(void)**
  A classical off by one error. Misusage of *strncpy* function. It is far better than *strcpy* since it performs boundary checks but does not check if the source can fit into the destination buffer. Consequently the length of *mystring* is greater than size of *myarr*. This can cause buffer and stack overflow.

```
1    char myarr[3];
2    len = strlen(mystring) - 1;
3    len = (len > 2) ? 2 : len;
4    strncpy(myarr, mystring, len);
5    myarr[2] = '\0';
```

Code fragment 6.1: Fix for example_bufferoverflow(void).

A check seen in Code fragment 6.1 or even better, replacing *strncpy* with a safe implementation like *strlcpy* fixes the vulnerability.

- **example_stacksmashing(void)**
  Similar vulnerability as in example_bufferoverflow(void). This vulnerability was inspired by Veracode's blog [Elliot M., 2013] and Ted Unangst's reply [Unangst T., 2013]. Code snippet is borrowed from Veracode's blog. Even being an experienced programmer and having the code reviewed by many, Ted had to learn the hard way that not the all the security measures work as expected between different hardware platforms. The operating system's built-in stack smashing protection was working correctly on Intel 386 platforms but did not work on AMD 64. Different alignment rules on AMD 64 hardware architecture caused the canary to lie further in the stack, and therefore permitted overflows. This is a good example of how hard it is to write secure code that works as intended on different platforms.

  The code includes an off by one error and the buffer overflow vulnerability. Scanf function sets the same size as in *myarr* initialization, hereby 32. Since the code uses char manipulator functions, one should be sure the last cell

(index of 31) is set to null, otherwise stack smashing can happen. Even better, functions that handle erroneous input correctly should be preferred, such as *sscanf*, *snprintf* or *fgets* depending on the parseable data.

- **example_nullcheck(void)**
  This function tries to open a file and print its contents to the screen. The user does not bother to check the return value of *fopen* and the program crashes.

  The function *fopen* returns a file descriptor for the opened file if it succeeds, null otherwise. Inserting *if (fp == NULL) err(1, NULL)* checks fixes the problem. In addition, fgets function does not guarantee null termination and hereby that must be taken care off by using sizeof function. An alternative way could be *strchr* function as shown in example_stacksmashing_fixed example.

- **void example_chrootandprivdrop()**
  In Freebsd, Netbsd and Openbsd *chroot* is implemented in a *vfs_syscall.c* file. All operating systems have implemented it differently. Examining userland usage of chroot reveals that there are also differences in calling the function. In history, there have been many known chroot vulnerabilities. Chroot's protective effect is dependent on operating system's implementation details. In short, not all chroot vulnerabilities work on all operating systems. Postfix [Postfix, 2013] was one of the first programs to utilize chroot and privilege separation to promote stronger security. Because of different implementations and for the portability, it is a good convention to call *chdir("/")* after a successful chroot call. Also, it is important to drop privileges after successful *chdir* execution. Otherwise user might be able to break out of the chroot.

  The problem in this example function is omitting the chdir call that can lead to an escape from chroot jail. Likewise omitting dropping the privileges may lead to the chroot escalation. Also, chroot should run with an UID (user ID) that is not used by any other daemon. This prevents debuggers from altering the outside processes. To fix the vulnerabilities, one should use chdir after a successful chroot call and *setuid* or similar call to drop the privileges.

- **void example_omiterrno()**
  Many functions use the *errno* variable to store data of erroneous functionality. This function changes a global *errno* value.

  The fix is simple. Introduce a local variable, for instance, *int savederrno = errno* at the beginning of the function. At the end of the function the global variable is updated: *errno = savederrno*.

- **void example_inputvalidation()**
  This function has an inappropriate input validation problem. With an ampersand in input user can set the intended command (first argument in here is dot) on the background process and run an arbitrary extra command on the latter argument (here *grep root /dev/passwd*). Grep root will fetch essential user account information regarding the root user, such as used shell, user ID and path of the home directory.

  Using functions *execv, execve* or *execl* helps to cope against the arbitrary command execution vulnerability since these functions implements input validation, at least on some level. In exec family functions, executable and parameters are given separately which makes malicious command injection harder.

7 Analyzing predefined flaws results

In this section, I will analyze the test flaw scan results and see how well the flaws were found. I will also pay attention to how errors are presented and if the analyzer gives a hint how to fix them. The complete transcript of the found flaws can be seen in Appendix D.

7.1 Compilers (Clang and GCC)

In many cases, output lines are indicating the same error, but express it differently. Both compilers detects basic issues fairly well. Issues are easy to parse due to consistent color usage and errors are pointed out from the context with a caret symbol. Redirecting ANSI colors causes garbled output if the receiver side does not support ANSI, hence colors are not shown in Appendix D. Issue description is terse but explains usually quite well what is wrong with the current code.

Both compilers give hints how the problem should be solved but does not necessarily expose the correct solution. In general, this is a useful since it helps unexperienced coders to spot the errors from code, although it may lead to a incorrect resolution because of a wrongly interpreted hint. For example, one can explicitly cast an int to an unsigned int to silence the warnings, although the variable should be able to store negative values too. Wrong casts are harder to debug since compilers will not yield warnings on them anymore. Almost all found issues were sane.

Clang yields a summary which indicates how many issues were found.

In general, both compilers help to catch easy flaws but do not perform further checks like control flow or data flow analysis. Neither of the two warns if insecure APIs are used.

7.2 Cppcheck

The result of cppcheck is extremely terse. It only found issues on example_bufferoverflow and example_nullcheck functions. Both issues are errors and should be fixed.

Cppcheck is the only one to provide fixing hints. The terse result suggested that the scan might not work properly. Fiddling with the switches and searching more information about extra checks did not change the output result to better.

Since both found flaws are real errors, it might be that cppcheck is concentrating on providing results that include real issues only, and uncertain issues are omitted. It is an ambitious goal, but almost impossible to achieve. With larger

projects this can be favourable, since that pays attention to fix the real problems, but at the same time it ignores other valuable flaws.

In comparison to the other results, the harsh approach seems to be pernicious. Too much optimization is bad for the results.

7.3 Flawfinder

The first impression is that flawfinder is rather verbose by default. It found errors from about half of the functions, including some of the fixed functions.

Flawfinder makes a good job explaining what the possible vulnerability is and how it could be exploited. It also gave useful porting hint that the *snprintf* function has flaky implementations on older systems. It was the only analyzer in addition to scan-build to note misuse of *chroot* function, although it failed to detect the proper use, exactly like scan-build. Announcing insecure APIs does not add any value for the result if the information is known beforehand and only makes the output more garbled. In addition, it lists every known insecure function but does not do any appropriate checks, like control flow or data analysis if the functions have been used correctly.

This indicates really poor functionality as it gives high false positive ratio and hides real flaws. Somewhat opposite what cppcheck does. Almost the same result can be achieved by using Unix's *grep* command against insecure functions and examining the output.

7.4 Scan-build

Scan-build found five issues. Two of them are for *chroot*. The latter one points to a fixed version and for some reason it cannot observe the existing *chdir* call. This led to perform some extra tests, where chdir call was set right after the chroot call. However, this did not remove the chroot false positives from the result. As mentioned in scan-build's introduction the manual page refers to certain tests with alpha prefix notation. It is unclear if alpha indicates its development state. This could explain the feeble chroot check results.

In Code fragment 7.1 line 10, scan-build reports unused variable, but that is a false positive because function *strtonum* assigns the value to *errstr*. Hereby the analyzer is not able to detect that. The remaining four issues are real and should be fixed. Scan-build's control flow analysis was the only one to spot the unreachable code, as seen in the Code fragment 7.2, line 11.

```
1   void
2   example_integeroverflow2_fixed(void)
3   {
4           long long int i = 0;
5           const char *errstr = NULL;
6
7           printf("example_integeroverflow2_fixed\n");
8
9           i = strtonum("1111111111111111", 0, UINT_MAX, &errstr);
10          if (errstr)
11                  errx(1, "number 1111111111111111 too big: %s", errstr);
12          printf("i = %llu\n", i);
13  }
```

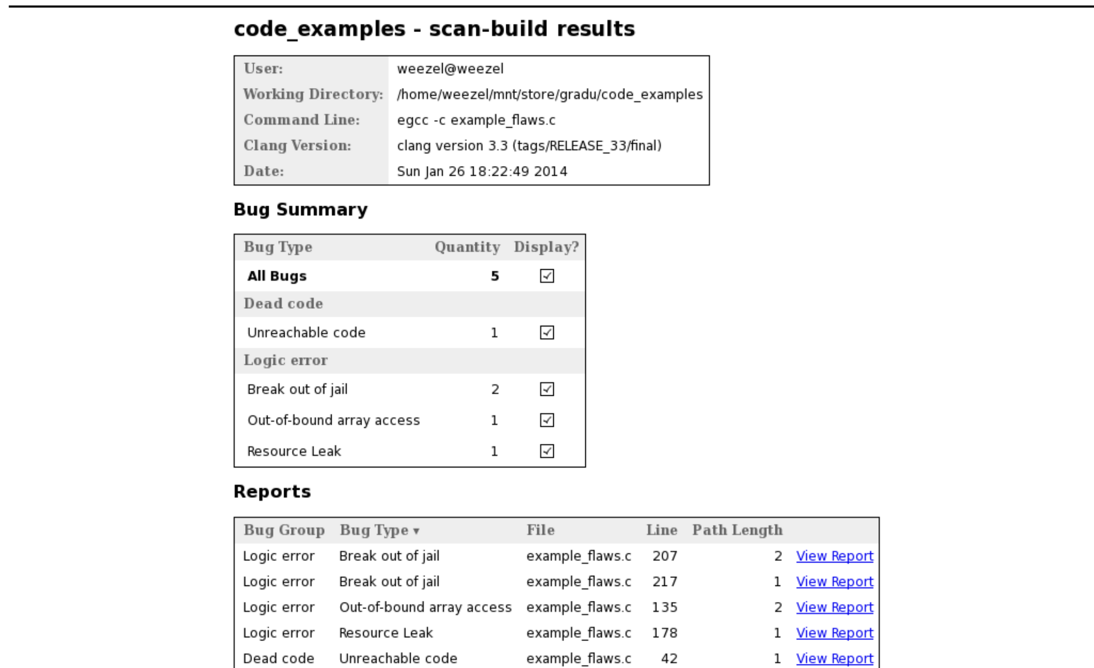
Code fragment 7.1: Predefined flaw: example_integeroverflow2_fixed function.

By conversion rules, *unsigned long int* will be converted to an *signed long int* form, since -1 cannot be represented as an unsigned int. Per C standard, it will be converted to *UINT_MAX*. In the end, UINT_MAX is always bigger than 10 and else is never reached.

```
1   void
2   example_signedness(void)
3   {
4           unsigned long int a = -10;
5           signed long int  b = 10;
6
7           printf("example_signedness\n");
8
9           printf("a = %lu\n", a);
10          printf("b = %ld\n", b);
11          if (a > b)
12                  printf("a is bigger\n");
13          else
14                  printf("b is bigger or equal\n");
15  }
```

Code fragment 7.2: Predefined flaw: example_signedness_fixed function.

Figure 7.1 shows the result page of the successful scanning and Figure 7.2 depicts how flaws are indicated in the detailed view.



code_examples - scan-build results

User:	weezel@weezel
Working Directory:	/home/weezel/mnt/store/gradu/code_examples
Command Line:	egcc -c example_flaws.c
Clang Version:	clang version 3.3 (tags/RELEASE_33/final)
Date:	Sun Jan 26 18:22:49 2014

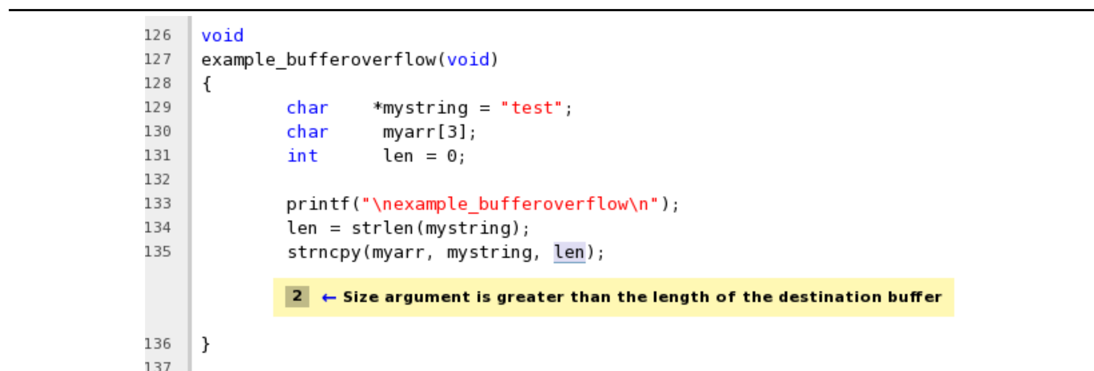
Bug Summary

Bug Type	Quantity	Display?
All Bugs	**5**	☑
Dead code		
Unreachable code	1	☑
Logic error		
Break out of jail	2	☑
Out-of-bound array access	1	☑
Resource Leak	1	☑

Reports

Bug Group	Bug Type ▾	File	Line	Path Length	
Logic error	Break out of jail	example_flaws.c	207	2	View Report
Logic error	Break out of jail	example_flaws.c	217	1	View Report
Logic error	Out-of-bound array access	example_flaws.c	135	2	View Report
Logic error	Resource Leak	example_flaws.c	178	1	View Report
Dead code	Unreachable code	example_flaws.c	42	1	View Report

Figure 7.1 Scan-build results page.



```
126  void
127  example_bufferoverflow(void)
128  {
129          char    *mystring = "test";
130          char     myarr[3];
131          int      len = 0;
132
133          printf("\nexample_bufferoverflow\n");
134          len = strlen(mystring);
135          strncpy(myarr, mystring, len);
                  2 ← Size argument is greater than the length of the destination buffer
136  }
137
```

Figure 7.2 Scan-build indicating buffer overflow flaw.

7.5 Splint

Splint's result is colossal, example_flaws.c is 299 lines of code, while splint's result has 129 lines, yielding 40 issues. The results also include detailed information of why the error message was given and gives a hint how it could be fixed.

Splint makes a good work parsing the function return types and notices if a settable variable has a different type. As seen in Code fragment 7.3, splint warns about differing return value.

```
1 example_flaws.c:131:27: Function strncpy expects arg 3 to be size_t gets int: len
```

Code fragment 7.3: Check of the function return value.

Checks seem not to be content aware and this explains a part of the false positives. Functions example_nullcheck and example_nullcheck_fixed deal with the file pointer *fp* and splint shows a possible issue in the example_nullcheck_fixed although it performs the null checks correctly. The return value of the *fclose* function can be ignored if it returns a non-zero result, since the control to the stream is already lost at this point. Like many other checkers, splint fails to check appropriate chroot usage.

There is also a hint how certain checks can be disabled when the issue appears for the first time. This would be better to be disabled by default since it makes the results harder to parse.

Splint reports the line number where the issue occurred, hence the function name does not add value into the results, although when integrated into an editor or IDE it may come handy.

7.6 Conclusion of predefined analyzing results

Analyzers can be divided into two discrete groups: the ones that add value to development process, and the others that report false issues and hence complicating the development process.

The best results regarding these test were given by Clang and scan-build combination. Splint produced a somewhat good result too, though there were many false positives. Using the rest of the analyzers can be discouraging due to their high false positive ratios.

Figure 7.3 has the overall results of the analyzers. In few cases, there are issues that trigger a warning in certain analyzers. Some of these are harmless but

still not false positives. In some situations, these could be counted as real bugs. Clang gives a warning when an int type variable is used as a parameter in strncpy function, for example. In this thesis, only absolute false positives are counted as false positives.

Test name	**clang + scan-build**	**cppcheck**	**flawfinder**	**gcc481**	**splint**
example_signedness	2			2	2
example_signedness_fixed					
example_integeroverflow1	2			1	
example_integeroverflow1_fixed					
example_integeroverflow2	1		1	1	1
example_integeroverflow2_fixed					4*
example_compare	1			1	1
example_compare_fixed					
example_bufferoverflow	3		3	2	2
example_bufferoverflow_fixed			1*		1*
example_stacksmashing	1	1	2	1	3
example_stacksmashing_fixed	1*	1*	1*		3*
example_nullcheck	1	1	2		4
example_nullcheck_fixed			2*		4*
example_chrootandprivdrop	1		1		
example_chrootandprivdrop_fixed	1*		1*		2*
example_omiterrno					1
example_omiterrno_fixed					
example_inputvalidation			3		2
example_inputvalidation_fixed			1*		3*
Total (falsepos/total)	2 / 9+5	1 / 3	6 / 18	8 / 0	17 / 42

Figure 7.3 Analyzers and found issues in predefined vulnerabilities. Issues are counted per function. Found issues are marked with numbers and false positives are marked with * character. Issues found from functions with _fixed postfix are counted as false positives. Empty cell means no issues found.

8 A real world example

This section examines a real world example of a *Tvheadend* [Tvhead, 2014] program which is a multimedia streaming server for Linux, supporting different kinds of DVB (Digital Video Broadcasting) technologies as input sources. Popularity in numbers on *Github* are: number of forks 286 at the time, starred 517 at the time and watch count 146. *Forks* indicates how many people have forked the repository to implement a certain functionality or create their own version from the program, for instance. *Watched* shows notifications in user's newsfeed (project's issues, comments and pull requests, etc.) *Starring* is similar to watch but it will not show notifications in newsfeed.

Running sloccount [Sloccount, 2014] in the *src* directory yields a result of 70951 source lines of code (SLOC). Analyzes were executed against e9a22d9 commit hash.

The tests were completed only in Linux environment since the software was initially designed for Linux and utilizes Linux specific APIs, *Video for Linux* (V4L) for example.

The choice of the real world example program was somewhat arbitrary, though some things were considered:

- It must be written purely in C
- Somewhat popular, though not from the top lists
- Cross-platform code, though this can be tolerated if otherwise applicable
- No static code analysis or annotation macros in source code since that means the code has been already reviewed

Github was chosen as a source, since it is one of the most popular open source sites on the Web. A common trend among newest server software seems to be that they are written in higher level languages. On the other hand, it can be a perception distortion: programs written in trending programming language get more attention in the media. The reason behind that could be for the higher level of abstraction, automatic memory handling and weak typing. Limiting search language to C reduced the top list results drastically. Nevertheless, there were still good candidates available. Cross-platform code would have been preferred though it was not mandatory.

I have read source code of some multimedia projects and remember excessive usage of bad APIs and unsafe functions. Attention was put on multimedia related

projects, preferable multimedia daemons in this regard. Multimedia related code needs to be optimized (e.g. codecs) and are usually feature rich due to support for different kind of file formats, streaming options, GUI customizations and so forth. The initial plan was to examine a music server software that I use daily, but unfortunately that was written in C++, and consequently not applicable.

The one intention was to use software that is not yet widely reviewed and hereby giving a higher understanding what kind of flaws statical analyzers can catch in the wild. With a few checked projects there were static code analysis related macros in the code and hereby superseded. On the other hand, it would have been interesting to see what kind of new flaws analyzers had found from the already analyzed code.

All issues cannot be commented on due to the complicated structure of tvheadend which would need familiarization. Some of the analyzers gave a list of many thousands issues, and analyzing each issue thoroughly is impossible in one thesis. Each tool's results are analyzed generally.

In order to explore the function calls more fluently, a tool named cscope [Cscope, 2012] and exuberant ctags [Exuberant Ctags, 2009] were used. These tools were immense helpful during the examination process.

Outputs could not be included in the appendix due to their excessive length.

8.1 Compilers (Clang and GCC)

The switches from the predefined vulnerabilities were used with minor tweaks, see Appendix E for detailed info. Some switches were removed from the Makefile since those dodged some important checks.

The most notable observation was that both compilers found lots of errors and signedness comparison issues. Clang performs slightly better than GCC. It is hard to estimate false positive ratio since the issues list was long.

The code has been written for Linux and it includes many GNU compiler specific extensions. With a pedantic switch enabled these issues are shown. There are no convincing reasons to use certain compiler specific annotations in code, since it makes code harder to read, harder to parse for analyzers and in some cases impossible to compile in different platforms.

It is hard to distinct how GCC's errors differ from Clang's errors with this much data. Regarding the summary, Clang outputs count of total errors whereas GCC does not. In total, Clang reports 7821 warnings. The original tvheadend Makefile also had a few Clang specific switches that were appended to a CFLAGS

variable during the compile time, mainly to disable certain checks. Some of the tvheadend's Makefile checks were probably disabled because functions are not implemented yet. It is hard to come with good reasons why the checks were disabled, especially when the code base was full of ignorance towards security. Clang related switches were enabled in this regard.

8.2 Cppcheck

The setup was really easy, though as a drawback the program was one of the slowest. On one file cppcheck could not perform parsing correctly. Seeing the file and the line number it is likely because of its macro parsing capabilities. Macro parsing capabilities were also a weak point for scan-build, which will be reviewed shortly.

The result was terse and the total number of issues found was 13. Nonetheless the result was easy to read and prudent. Cppcheck was the only one to find realloc issues in *hstmsg.c* file.

Moreover, many of the reported issues were real, hereby false positive ratio is good but still, to emphasize, terse. Total 13 issues out of over 70 000 SLOC can give a wrong impression about the code quality.

8.3 Flawfinder

The only difference to the example code review is that Flawfinder output was set to HTML so the result could be reviewed in a browser. However, the result was still hard to parse for the human eye. In some other code base, Flawfinder would not be as successful as it was now. Wrong API (strcpy for example) usage is easy to spot with "buffer" tagged lines, though not every buffer tagged line is a vulnerability. Hereby, it is good for catching unsafe APIs but otherwise does not give that much value as an analyzer.

8.4 Scan-build

Scan-build found 317 issues, albeit there were parsing errors during the analysis process. Enabled checks were same to the predefined flaws section, excluding the chroot check since that was proven to be non-functional. The issues count by category is shown in Figure 8.1.

Closer examination of a few randomly chosen issues shows that in file *intl-conv.c* line 156, *alloca* reserved resource is freed, although it should not. That

Bug type	Quantity
API	2
Dead code	36
Dead store	6
Logic error	183
Memory error	37
Security	38
Unix API	9
Unix Stream API error	6
Total	**317**

Figure 8.1 Scan-build analysis results of tvheadend.

causes undefined behaviour and at worst, triggers a stack smashing overflow. In addition, to the scan-build report, manual page of GNU Linux reminds not to free alloca reserved resources. In *imagecache.c* file, a resource leak happens in *imagecache_image_fetch* function because *FILE *fp* variable is never closed before the function returns.

Overall all the extra checks tend to be vain or in beta phase. Opposite to cppcheck, scan-build is excessively verbose with all the checks enabled. False positive ratio is high, though using default settings the situation normalizes. With default settings scan-build yields 89 issues and the count of false positives is reduced.

8.5 Splint

Preparing of Splint started by editing the Makefile of tvheadend, see Appendices E and F for a detailed information. Splint could not find appropriate header files and refused to continue. A workaround is to iterate every C file and execute Splint for each. This can be achieved with a simple shell script shown in Appendix F. When a project gets bigger and uses many programming languages the script maintenance becomes impractical.

Analyzing the result files reveals that preprocessing errors were still around. The total number of files that could not be parsed was 299 out of 358. The manual page of Splint did not mention debug or verbose modes, and that makes it is hard to find why files could not be parsed.

Overall, it was really hard to parse real errors from the results since it was filled

with missing header errors. It is not meaningful to measure speed performance nor false positive ratio since so many files were skipped.

It is tedious and awkward process to integrate Splint into a project, not to mention it failed to parse over 80% of the files.

8.6 Manual code review

Performing a manual code review on realloc function calls reveals that realloc was used incorrectly overall. In one file, *htsbuf.c*, realloc was used corretly. Comments have references to *glibc* library which could explain the exception.

Code fragment 8.1 is taken from a file *httpc.c*. If the realloc function call fails it can leak memory and cause indeterministic behaviour. Code fragment 8.2 addresses these problems.

```
1  if (ssl->rbio_pos + len > ssl->rbio_size) {
2    ssl->rbio_buf = realloc(ssl->rbio_buf, ssl->rbio_pos + len);
3    ssl->rbio_size += len;
4  }
```

Code fragment 8.1: Using realloc incorrectly.

```
1  unsigned char *buf;
2
3  if (ssl->rbio_pos + len > ssl->rbio_size) {
4    /* If realloc returns NULL, ssl->rbio_buf is still allocated */
5    if ((buf = realloc(ssl->rbio_buf, ssl->rbio_pos + len)) == NULL) {
6      free(ssl->rbio_buf); /* Without these leak(s) can happen */
7      ssl->rbio_buf = NULL; /* Without these leak(s) can happen */
8      ssl->rbio_size = 0; /* Without these leak(s) can happen */
9      return NULL;
10   }
11   ssl->rbio_buf = buf;
12   memcpy(ssl->rbio_buf + ssl->rbio_pos, rbuf + (len - r), len);
13   ssl->rbio_size += len;
14 }
```

Code fragment 8.2: More eligible way to use realloc.

Another quick check for malloc and calloc function calls reveals that those return values are not checked either. In addition, all memory allocation function's size values should be checked for underflows and overflows.

This would be a good starting point for the tvheadend team to practice manual peer code reviews and get some security oriented developers in the team.

8.7 Conclusion of the real world example

Unsafe functions are used all over in the code. Using unsafe APIs and function calls, strcpy for example, tells something about the code base. There are barely return value checks. For instance, the Code fragment 8.1 depicts the realloc misuse by omitting a null check (return value). Searching usage of *strcpy* and *strcat* in the source directory yields 34 results. Using a semantic patcing tool like Coccinelle [Coccinelle, 2014] automates the conversion workload. Most of the found issues are quick and easy to fix and there is no real excuse to omit the amendments.

Learning and writing the C language without ever hearing a precaution about buffer overflow vulnerabilities sounds a bit unlikely. Especially when the buffer overflow vulnerabilities have been in the CWE/SANS Top 25 list for decades [CWE Top 25, 2011] and covered by the most of the learning materials.

It would have been interesting to analyze tvheadend code base with commercial analyzers such as *Coverity*, *Klokwork* and *Fortify* even though the initial idea was to use freely available open source tools. One factor could have been to compare whether the commercial ones are worth of money.

If the intention is to write secure code one should adopt the above introduced design principles as a part of the coding process. Using privilege separation with chroot helps to mitigate whole server compromise. Favoring design principles helps to cope with badly designed APIs, reduces the attack surface and primarily makes code easier to understand. Chrooting the tvheadend to a distinct directory, running it as a dedicated user, dropping extra privileges and separating client access from the server would make it way more safe to use.

Nevertheless, even if good design principles would have been adopted it is important to use safe APIs. Unsafe system calls can dilute the advantage gained by good design principles. It is important to analyze code one or more analyzers before the release to catch at least the most obvious unsafe system calls and API usages.

As shown by the examples, static code analyzers are able to catch bugs but

it always needs user's intervention. Unsafe API usage and omitted error checks gives an impression that tvheadend developers have barely taken any security measures into account during the development.

9 Conclusion and future studies

Completely secure systems do not exist. Humans are and will always be the weakest link what comes to the computer security. Humans write the software, develop the hardware and use the computers – appropriately or inappropriately. Well designed security environment does not guarantee protection against all plausible flaws that exist but makes system penetration and exploitation harder. Even the safest systems cannot be left without updates after the initial setup has been completed in a safe manner. Security is a race against time where the only hope is incompetence of a malicious attacker. Many of the introduced methods offer a safer way to execute programs, hide classified information, sustain integrity or make service fail on a safe manner.

As a conclusion to the language based security, several badly written APIs exist and are used daily. Also as proven by the thesis, many of them are ambiguous and hard to use consistently. A leap in the development of security is to substitute all unsafe functions with boundary-aware functions and use APIs that have consistent interface. Rather than implementing programs to tighten up application's security, one should consider how the original program was written, refactor if needed and adapt secure principles shown in this thesis. Using the design patterns and well known security measures will help to port programs to different platforms. Since it takes only a few extra steps and a little more pondering to make programs considerably harder to exploit, there is no excuse to take a shortcut. Advantages are be obvious and also answers to the first research question.

Nevertheless, even the good design principles can be diluted by using unsafe APIs. Luckily static code analysis is able to point out obvious flaws before they are being exploited in the wild. Nowadays static code analyzers are getting more intelligent and context awareness is getting better. Still, not all the issues are necessarily flaws as can be seen by the examples. Every scan result needs a thorough investigation before doing the fixes. This also answers the second research question. A plain compiler makes a good job of catching bugs, however it does not remove the need for static code analysis and manual reviews.

If the security is the main concern, performance overhead is inevitable, since values must be checked against the different safety limits. In a modern computer, regression is almost indistinguishable so it is not a valid excuse to omit checks. Good security is multilayered which does not rely on one security measurement.

Using annotations in static code analysis would be interesting for the future

studies. This would give more sophisticated results for many analyzers. That would also mean more work since there is no common syntax among the analyzers.

It is not a rare thing in open source projects that code is not analyzed or reviewed by a third party even if it is enjoying large popularity. Way too many expect someone else to do the job. On commercial side the problem is even worse, since the programs are usually distributed as a precompiled binary that cannot be debugged or analyzed easily. Vendor's word is the only promise about the code quality. Can the vendor be trusted and is analysis review done by the competent developer? These questions should be asked when choosing secure software.

References

[Advances in OpenBSD, 2013] Advances in OpenBSD. <`http://www.openbsd.org/papers/csw03/index.html`> (accessed: 19.12.2011).

[C99 standard, 2007] Final version of the C99 standard. <`http://www.open-std.org/jtc1/sc22/WG14/www/docs/n1256.pdf`> (accessed 27.01.2014).

[Cauldwell P., 2008] Cauldwell P. *Using people, tools, and processes to build successful software* Wiley, Indianapolis, Indiana, 2008.

[Clang Analyzer, 2013] Clang Static Analyzer. <`http://clang-analyzer.llvm.org`> (accessed 10.10.2013).

[Clang Scanbuild, 2014] Clang Analyzer - scan-build: running the analyzer from the command line. <`http://clang-analyzer.llvm.org/scan-build.html`> (accessed 25.01.2014).

[Coccinelle, 2014] Coccinelle: A Program Matching and Transformation Tool for Systems Code. `http://coccinelle.lip6.fr/` (accessed 13.08.2014).

[Cooper K. D. et al., 2013] Cooper K. D. et al. Iterative Data-flow Analysis, Revisited *Rise University* Houston, Texas, USA. <`http://www.cs.rice.edu/~harv/my_papers/worklist.pdf`> (accessed 26.10.2013).

[Corbet J., 2012] Corbet J. LWN.net. <`http://lwn.net/Articles/493599/`> (accessed 01.12.2013).

[Coverity, 2013] Coverity Scan - Static Analysis. <`https://scan.coverity.com/faq#what-types-of-issues-tool-find`> (accessed 25.09.2013).

[cppcheck, 2013] cppcheck. <`http://cppcheck.wiki.sourceforge.net/`> (accessed 12.10.2013).

[Cscope, 2012] Cscope. Cscope Home Page. <`http://cscope.sourceforge.net/`> (accessed 02.06.2014).

[CWE-20, 2013] CWE-20: Improper Input Validation. <`http://cwe.mitre.org/data/definitions/20.html`> (accessed 31.11.2013).

[CWE-120, 2013] CWE-120: Buffer Copy without Checking Size of Input ('Classic Buffer Overflow'). <http://cwe.mitre.org/data/definitions/120.html> (accessed 30.11.2013).

[CWE-227, 2013] CWE-227: Improper Fulfillment of API Contract ('API Abuse'). <http://cwe.mitre.org/data/definitions/227.html> (accessed 30.11.2013).

[CWE-388, 2013] CWE-388: Error Handling. <http://cwe.mitre.org/data/definitions/388.html> (accessed 30.11.2013).

[CWE-391, 2013] CWE-391: Unchecked Error Condition. <http://cwe.mitre.org/data/definitions/391.html> (accessed 30.11.2013).

[CWE-457, 2013] CWE-457: Use of Uninitialized Variable. <http://cwe.mitre.org/data/definitions/457.html> (accessed 30.11.2013).

[CWE-569, 2013] CWE-569: Expression Issues. <http://cwe.mitre.org/data/definitions/569.html> (accessed 30.11.2013).

[CWE-570, 2013] CWE-570: Expression is Always False. <http://cwe.mitre.org/data/definitions/570.html> (accessed 31.11.2013).

[CWE-622, 2013] CWE-662: Improper Synchronization. <http://cwe.mitre.org/data/definitions/622.html> (accessed 31.11.2013).

[CWE-658, 2014] CWE-658: Weaknesses in Software Written in C. <http://cwe.mitre.org/data/lists/658.html> (accessed 23.01.2014).

[CWE-670, 2013] CWE-670: Always-Incorrect Control Flow Implementation. <http://cwe.mitre.org/data/definitions/670.html> (accessed 30.11.2013).

[CWE Top 25, 2011] CWE/SANS Top 25 Most Dangerous Software Errors. <http://cwe.mitre.org/top25/#Listing> (accessed 30.11.2013).

[de Raadt, 2011] de Raadt T., Discussion on buffer overflow prevention issues. <http://openbsd.org/papers/csw03/mgp00015.html> (accessed 12.12.2011).

[Elliot M., 2013] Elliot M. A tale of two compilers - Veracode security blog. <http://www.veracode.com/blog/2013/11/a-tale-of-two-compilers/> (accessed: 30.12.2011).

[Ericksson J., 2003] Erickson J. *Hacking: The Art of Exploitation*, No Starch Press Inc., San Francisco, CA, 2003.

[Evans D. and Larochelle D., 2002] Evans D. and Larochelle D., Improving Security Using Extensible Lightweight Static Analysis, *IEEE Software*, **19** (2002), 42-51.

[Exuberant Ctags, 2009] Exuberant Ctags. <http://ctags.sourceforge.net/> (accessed 02.06.2014).

[Flawfinder, 2013] Flawfinder. <http://www.dwheeler.com/flawfinder> (accessed 08.02.2013).

[FreeBSD malloc, 2014] malloc, calloc, realloc, free, reallocf, malloc_usable_size – general purpose memory allocation functions <http://www.freebsd.org/cgi/man.cgi?query=malloc.conf&apropos=0&sektion=0&manpath=FreeBSD+9.2-RELEASE&arch=default&format=html> (accessed 20.01.2014).

[GCC, 2013] GCC. The GNU Compiler Collection. <http://gcc.gnu.org/> (accessed 26.09.2013).

[GCC Bugzilla, 2007] GCC Bugzilla. Bug 30785 - Test to null pointer optimised away at -O2. <http://gcc.gnu.org/bugzilla/show_bug.cgi?id=30785> (accessed 23.03.2014).

[Gleick J., 2010] Gleick J. A bug and a crash - Sometimes a bug is more than a nuisance. <http://www.around.com/ariane.html> (accessed: 02.10.2013).

[IBM Research, 2001] IBM Research. 2001. GCC extension for protecting applications from stack-smashing attacks. <http://www.trl.ibm.com/projects/security/ssp/> (accessed 10.11.2011).

[ISO 8601, 2013] International Organization for Standardization. Date and time format - ISO 8601. <http://www.iso.org/iso/iso8601> (accessed 12.09.2013).

[Hass A. M. J., 2008] Hass A. M. J., *Guide to Advanced Software Testing,* Artech House, Norwood, Massachusetts, 2008.

[Lévy J-J, 2010] Lévy J-J. 2010. Un petit bogue, un grand boum! Inria Microsoft research. <http://moscova.inria.fr/~levy/talks/10enslongo/enslongo.pdf> (accessed: 02.10.2013).

[LLVM, 2013] The LLVM compiler infrastructure. <http://www.llvm.org/> (accessed 10.10.2013).

[Mastropaolo M., 2005] Mastropaolo M., Buffer overflow attacks bypassing DEP (NX/XD bits) - part 2 : Code injection. <http://www.mastropaolo.com/2005/06/05/buffer-overflow-attacks-bypassing-dep-nxxd-bits-part-2-code-injection/> (accessed 14.12.2011).

[Miller T. C. and de Raadt T., 1996] Miller T. C. and de Raadt T. Strlcpy and strlcat - Consistent, Safe, String Copy and Concatenation. In: *USENIX Conference -99*. Monterey, California. <http://openbsd.org/papers/strlcpy-paper.pdf> (accessed 02.10.2011).

[Monit, 2013] Monit. M/Monit <http://mmonit.com/> (accessed 19.09.2013).

[Nagpal, 2009] Nagpal N. *Unix and Shell Programming*, Global Media, Lucknow, India, 2009.

[Null Deference - OWASP, 2013] Null deference - OWASP. <https://www.owasp.org/index.php/Null_Dereference> (accessed 30.10.2013).

[OpenBSD, 2013] OpenBSD operating system. <http://www.openbsd.org> (accessed: 04.09.2013).

[OpenBSD malloc, 2013] Malloc, calloc, realloc, free, cfree - memory allocation and deallocation. <http://www.openbsd.org/cgi-bin/man.cgi?query=malloc&apropos=0&sektion=0&manpath=OpenBSD+Current&arch=i386&format=html> (accessed 22.08.2013).

[Poll E., 2011a] Poll E. 2011a. Software security lecture notes. Lecture of design principles. The Kerckhoffs institute, Radboud. Nijmegen. <http://www.cs.ru.nl/E.Poll/ss/> (accessed 17.02.2012).

[Poll E., 2011b] Poll E. 2011b. Software security lecture notes. Lecture of language based security 1. The Kerckhoffs institute. Radboud. Nijmegen. <http://www.cs.ru.nl/E.Poll/ss/> (accessed 17.02.2012).

[Poll E., 2011c] Poll E. 2011c. Software security lecture notes. Lecture of input validation. The Kerckhoffs institute. Radboud. Nijmegen. <http://www.cs.ru.nl/E.Poll/ss/> (accessed 17.02.2012).

[Poll E., 2011d] Poll E. 2011d. Software security lecture notes. Lecture of program analysis with PREfast & SAL. The Kerckhoffs institute. Radboud. Nijmegen. <http://www.cs.ru.nl/E.Poll/ss/> (accessed 17.02.2012).

[Provos et al, 2003] Provos N., Fiedl M., Honeyman P. *Preventing Privilege Escalation.* In: *Proceedings of the 12th USENIX Security Symposium.* Washington, DC, 2003.

[Postfix, 2013] The Postfix Home Page. <http://www.postfix.org> (accessed 14.12.2013).

[Provos et al, 2010] Puffy at work - Code right and secure, The OpenBDS way. <http://quigon.bsws.de/papers/2010/bsdcan/mgp00046.html> (accessed 14.11.2011).

[Resource Leak - OWASP, 2013] Resource Leak - OWASP. <https://www.owasp.org/index.php/Unreleased_Resource> (accessed 30.10.2013).

[Sloccount, 2014] SLOCCount. <http://www.dwheeler.com/sloccount/> (accessed 02.06.2014).

[Splint, 2013] Splint. <http://www.splint.org> (accessed 08.02.2013).

[Splint manual, 2013] Splint manual. <http://www.splint.org/manual/manual.html> (accessed 27.10.2013).

[SQL Injection - OWASP, 2013] SQL Injection - OWASP. <https://www.owasp.org/index.php/SQL_Injection> (accessed 18.07.2013).

[Stuttard, D. and Pinto, M., 2008] Stuttard, D. and Pinto, M. *Web Application Hackers Handbook: Discovering and Exploiting Security Flaws,* Wiley, New Jersey, 2008.

[Supervisor, 2013] Supervisor: A Process Control System. <http://supervisord.org/>, (accessed 19.09.2013).

[TCP Wrappers, 2011] TCP Wrappers. <http://www.softpanorama.org/Net/Network_security/TCP_wrappers/index.shtml> (accessed 18.10.2011).

[The Free Dictionary, 2012] The Free Dictionary. <http://encyclopedia2.thefreedictionary.com/Keep+It+Simple%2c+Stupid>, (accessed 20.12.2012).

[The Open Group, 2011] The open group base specifications issue 6. <http://pubs.opengroup.org/onlinepubs/009604599/functions/strncat.html> (accessed: 11.12.2011).

[Tremplay J-P and Sorenson, 2008] Tremblay J.-P. and Sorenson P. G., *Theory and Practice of Compiler Writing*, Global Media, Hyderabad, India, 2008.

[Tvhead, 2014] Tvhead. Tvheadend is a TV streaming server for Linux supporting DVB-S, DVB-S2, DVB-C, DVB-T, ATSC, IPTV, and Analog video (V4L) as input sources. <https://tvheadend.org/> (accessed 02.06.2014).

[Unangst T., 2013] Unangst T. is your stack protector working? <http://www.tedunangst.com/flak/post/my-stack-protector-wasnt-working> (accessed 22.12.2013).

[Valgrind, 2013] Valgrind. About Valgrind <http://www.valgrind.org/info/about.html> (accessed 26.09.2013).

[Wilander and Kamkar, 2000] Wilander J. and Kamkar M., A Comparison of Publicly Available Tools for Dynamic Buffer Overflow Prevention, Linköpings universitet.

[Wilton and Colby, 2005] Wilton P. and Colby, J. *Beginning SQL*, Wiley, New Jersey, 2005.

[Wu-FTP, 2011] WU-FTPD Development Group. <http://wu-ftpd.therockgarden.ca> (accessed: 02.12.2011).

A Java JDBC Stack trace

```
com.mysql.jdbc.exceptions.MySQLSyntaxErrorException: You have an error in your
SQL syntax; check the manual that corresponds to your MySQL server version for
the right syntax to use near 'and afdeling.provincie in (0,3) order by
afdeling.sort' at line 3
        at com.mysql.jdbc.SQLError.createSQLException(SQLError.java:936)
        at com.mysql.jdbc.MysqlIO.checkErrorPacket(MysqlIO.java:2941)
        at com.mysql.jdbc.MysqlIO.sendCommand(MysqlIO.java:1623)
        at com.mysql.jdbc.MysqlIO.sqlQueryDirect(MysqlIO.java:1715)
        at com.mysql.jdbc.Connection.execSQL(Connection.java:3243)
        at com.mysql.jdbc.Connection.execSQL(Connection.java:3172)
        at com.mysql.jdbc.Statement.execute(Statement.java:706)
        at com.mysql.jdbc.Statement.execute(Statement.java:783)
        at coldfusion.server.j2ee.sql.JRunStatement.execute(JRunStatement.java:348)
        at coldfusion.sql.Executive.executeQuery(Executive.java:1210)
        at coldfusion.sql.Executive.executeQuery(Executive.java:1008)
        at coldfusion.sql.Executive.executeQuery(Executive.java:939)
        at coldfusion.sql.SqlImpl.execute(SqlImpl.java:325)
        at coldfusion.tagext.sql.QueryTag.executeQuery(QueryTag.java:831)
        at coldfusion.tagext.sql.QueryTag.doEndTag(QueryTag.java:521)
        at cfindex2ecfm893406114.runPage(D:\websites\kicker\content\Uitslagen\index.cfm:32)
        at coldfusion.runtime.CfJspPage.invoke(CfJspPage.java:192)
        at coldfusion.tagext.lang.IncludeTag.doStartTag(IncludeTag.java:366)
        at coldfusion.filter.CfincludeFilter.invoke(CfincludeFilter.java:65)
        at coldfusion.filter.ApplicationFilter.invoke(ApplicationFilter.java:279)
        at coldfusion.filter.RequestMonitorFilter.invoke(RequestMonitorFilter.java:48)
        at coldfusion.filter.MonitoringFilter.invoke(MonitoringFilter.java:40)
        at coldfusion.filter.PathFilter.invoke(PathFilter.java:86)
        at coldfusion.filter.ExceptionFilter.invoke(ExceptionFilter.java:70)
        at coldfusion.filter.ClientScopePersistenceFilter.invoke(ClientScopePersistenceFilter.java:28)
        at coldfusion.filter.BrowserFilter.invoke(BrowserFilter.java:38)
        at coldfusion.filter.NoCacheFilter.invoke(NoCacheFilter.java:46)
        at coldfusion.filter.GlobalsFilter.invoke(GlobalsFilter.java:38)
        at coldfusion.filter.DatasourceFilter.invoke(DatasourceFilter.java:22)
        at coldfusion.CfmServlet.service(CfmServlet.java:175)
        at coldfusion.bootstrap.BootstrapServlet.service(BootstrapServlet.java:89)
        at jrun.servlet.FilterChain.doFilter(FilterChain.java:86)
        at coldfusion.monitor.event.MonitoringServletFilter.doFilter(MonitoringServletFilter.java:42)
        at coldfusion.bootstrap.BootstrapFilter.doFilter(BootstrapFilter.java:46)
        at jrun.servlet.FilterChain.doFilter(FilterChain.java:94)
        at jrun.servlet.FilterChain.service(FilterChain.java:101)
        at jrun.servlet.ServletInvoker.invoke(ServletInvoker.java:106)
        at jrun.servlet.JRunInvokerChain.invokeNext(JRunInvokerChain.java:42)
        at jrun.servlet.JRunRequestDispatcher.invoke(JRunRequestDispatcher.java:284)
        at jrun.servlet.ServletEngineService.dispatch(ServletEngineService.java:543)
        at jrun.servlet.jrpp.JRunProxyService.invokeRunnable(JRunProxyService.java:203)
        at jrunx.scheduler.ThreadPool$DownstreamMetrics.invokeRunnable(ThreadPool.java:320)
        at jrunx.scheduler.ThreadPool$ThreadThrottle.invokeRunnable(ThreadPool.java:428)
        at jrunx.scheduler.ThreadPool$UpstreamMetrics.invokeRunnable(ThreadPool.java:266)
        at jrunx.scheduler.WorkerThread.run(WorkerThread.java:66)
```

B EXAMPLE_FLAWS.C

```c
#include <err.h>
#include <errno.h>
#include <math.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#if defined __linux__
#include <sys/types.h>
#include <limits.h>
#endif

void
example_signedness(void)
{
	unsigned long int a = -10;
	signed long int  b = 10;

	printf("example_signedness\n");

	printf("a = %lu\n", a);
	printf("b = %ld\n", b);
	if (a > b)
		printf("a is bigger\n");
	else
		printf("b is bigger or equal\n");
}

void
example_signedness_fixed(signed long b)
{
	signed long int a = -10;

	printf("example_signedness_fixed\n");

	printf("a = %ld\n", a);
	printf("b = %ld\n", b);
	if (a > b)
		printf("a is bigger\n");
	else
```

```
43                      printf("b is bigger or equal\n");
44  }
45
46  void
47  example_integeroverflow1(void)
48  {
49          signed int i = 0xbadbadbad;
50
51          printf("example_integeroverflow1\n");
52
53          printf("i = %d\n", i);
54  }
55
56  void
57  example_integeroverflow1_fixed(void)
58  {
59          unsigned long int i = 0xbadbadbad;
60
61          printf("example_integeroverflow1_fixed\n");
62
63          printf("i = %lu\n", i);
64  }
65
66  void
67  example_integeroverflow2(void)
68  {
69          int i = 0;
70
71          printf("example_integeroverflow2\n");
72
73          i = atol("1111111111111111");
74          printf("i = %d\n", i);
75  }
76
77  void
78  example_integeroverflow2_fixed(void)
79  {
80          long long int i = 0;
81          const char *errstr = NULL;
82
83          printf("example_integeroverflow2_fixed\n");
84
85          i = strtonum("1111111111111111", 0, UINT_MAX, &errstr);
86          if (errstr)
```

```
87                      errx(1, "number 1111111111111111 too big: %s", errstr);
88              printf("i = %llu\n", i);
89      }
90
91      void
92      example_compare(void)
93      {
94              double a = 0.00000001;
95              double b = 0.00000002;
96
97              printf("\nexample_compare\n");
98
99              printf("a = %f\n", a);
100             printf("b = %f\n", b);
101
102             if (a == b)
103                     printf("a and b are equal\n");
104             else
105                     printf("a and b not equal\n");
106     }
107
108     void
109     example_compare_fixed(void)
110     {
111             double a = 0.00000001;
112             double b = 0.00000002;
113
114             printf("\nexample_compare_fixed\n");
115
116             printf("a = %f\n", a);
117             printf("b = %f\n", b);
118
119             if (fabs(a - b) < 1.0E-4)
120                     printf("a and b are equal\n");
121             else
122                     printf("a and b not equal\n");
123     }
124
125     void
126     example_bufferoverflow(void)
127     {
128             char *mystring = "test";
129             char myarr[3];
130             int len = 0;
```

```
131
132         printf("\nexample_bufferoverflow\n");
133         len = strlen(mystring);
134         strncpy(myarr, mystring, len);
135 }
136
137 void
138 example_bufferoverflow_fixed(void)
139 {
140         char *mystring = "test";
141         char myarr[3];
142
143         printf("\nexample_bufferoverflow_fixed\n");
144         strlcpy(myarr, mystring, sizeof(myarr));
145 }
146
147 // Borrowed from Veracode's blog, written by Elliot M.
148 void
149 example_stacksmashing(void)
150 {
151         int  i = 0xabad1dea;
152         char  myarr[32];
153
154         printf("\nexample_stackoverflow\n");
155         scanf("%32s", myarr);
156         printf("%s 0x%x\n", myarr, i);
157 }
158
159 void
160 example_stacksmashing_fixed(void)
161 {
162         unsigned int  i = 0xabad1dea;
163         char   myarr[32];
164         char   *p = NULL;
165
166         printf("\nexample_stackoverflow_fixed\n");
167         if (fgets(myarr, sizeof(myarr), stdin) != NULL) {
168                 if ((p = strchr(myarr, '\n')) == NULL)
169                         err(1, "line too long?, aborting");
170         }
171         *p = '\0';
172         printf("%s 0x%x\n", myarr, i);
173 }
174
```

```
175 void
176 example_nullcheck(void)
177 {
178         FILE *fp;
179         char linebuf[256];
180 
181         printf("\nexample_nullcheck\n");
182         fp = fopen("nonexistent", "r");
183         fgets(linebuf, sizeof(linebuf), fp);
184         printf("linebuf: %s\n", linebuf);
185 }
186 
187 void
188 example_nullcheck_fixed(void)
189 {
190         FILE *fp;
191         char linebuf[256];
192 
193         printf("\nexample_nullcheck_fixed\n");
194         if ((fp = fopen("nonexistent", "r")) == NULL)
195                 err(1, NULL);
196         if (fgets(linebuf, sizeof(linebuf), fp) == NULL)
197                 goto error;
198         linebuf[sizeof(linebuf) - 1] = '\0';
199         printf("linebuf: %s\n", linebuf);
200 
201 error:
202         if (fp)
203                 fclose(fp);
204 }
205 
206 void
207 example_chrootandprivdrop(void)
208 {
209         printf("\nexample_chroot_as_user\n");
210         /* In a real application we must exit if the chroot fails. Here we use
211          * warn() function just for the indication. */
212         if (chroot("/tmp") == -1)
213                 warn("cannot chroot to /tmp");
214 }
215 
216 void
217 example_chrootandprivdrop_fixed(void)
218 {
```

```
219         gid_t newgid = 1000;
220
221         printf("\nexample_chroot_as_user_fixed\n");
222         if (chroot("/tmp") != 0 || chdir("/") != 0)
223                 err(1, "%s", "/tmp");
224          /* Important to drop extra privileges since otherwise user could
225           * escape the restricted environment. */
226         if (setgroups(1, &newgid) ||
227             setresgid(newgid, newgid, newgid) ||
228             setresuid(newgid, newgid, newgid))
229                 err(1, "cannot drop privileges");
230 }
231
232 void
233 example_omiterrno(int signal)
234 {
235         switch (signal) {
236         case SIGPIPE:
237                 printf("broken pipe\n");
238                 break;
239         default:
240                 return;
241         }
242 }
243
244 void
245 example_omiterrno_fixed(int signal)
246 {
247         int olderrno;
248
249         olderrno = errno;
250
251         switch (signal) {
252         case SIGPIPE:
253                 if (unlink("non_existant_file") == -1)
254                         fprintf(stderr, "error: %s\n", strerror(errno));
255                 printf("broken pipe\n");
256                 break;
257         default:
258                 return;
259         }
260
261         errno = olderrno;
262 }
```

```
263
264 void
265 example_inputvalidation(void)
266 {
267         char  cmd[1024];
268         char  *userinput = ". & grep root /etc/passwd";
269
270         snprintf(cmd, sizeof(cmd), "ls %s", userinput);
271         system(cmd);
272 }
273
274 void
275 example_inputvalidation_fixed(void)
276 {
277         char  *userinput = ". & grep root /etc/passwd";
278         char  *cmd[] = {"ls", userinput, NULL};
279         pid_t  pid;
280
281         if ((pid = fork()) < 0)
282                 err(1, "fork failed");
283         else if (pid == 0) {
284                 execv("/bin/ls", cmd);
285                 _exit(127);
286         }
287 }
288
289 int
290 main(void)
291 {
292         signal(SIGPIPE, example_omiterrno);
293         example_signedness();
294         example_integeroverflow1();
295         example_integeroverflow2();
296         example_compare();
297         example_stacksmashing();
298         example_bufferoverflow();
299         example_nullcheck();
300         example_chrootandprivdrop();
301         example_inputvalidation();
302
303         return 0;
304 }
```

C Makefile

```
CFLAGS  = -std=c99 -pedantic -Wall -fsyntax-only
CFLAGS  += -Wformat=2 -Wunused-parameter -Wsystem-headers \
                    -Wfloat-equal -Wshadow -Wpointer-arith -Wtype-limits \
                    -Wcast-qual -Wcast-align -Wconversion -Wempty-body \
                    -Wenum-compare -Wsign-compare -Wsizeof-pointer-memaccess \
                    -Waddress -Wredundant-decls -Wpointer-sign \
                    -Wstack-protector -fstack-protector -Woverlength-strings
OUTPUT  = codechecks.log
OUTPUT_HTML = codechecks_html

.PHONY: all clean

all: clean scanbuild cppcheck flawfinder gcc482 splint

cppcheck:
	@echo "\n### $$(cppcheck --version) ###" >> ${OUTPUT} 2>&1
	@echo "###" >> ${OUTPUT}
	cppcheck -v --std=c99 example_flaws.c >> ${OUTPUT} 2>&1
flawfinder:
	@echo "\n### flawdinfer $$(flawfinder --version) ###" >> ${OUTPUT} 2>&1
	@echo "###" >> ${OUTPUT}
	flawfinder -n -S example_flaws.c >> ${OUTPUT} 2>&1
gcc482:
	@echo "\n### $$(egcc --version |head -1) ###" >> ${OUTPUT} 2>&1
	@echo "###" >> ${OUTPUT}
	egcc $(CFLAGS) example_flaws.c >> ${OUTPUT} 2>&1
scanbuild:
	@echo "### $$(clang -v 2>&1 |head -1) and scan-build ###" >> ${OUTPUT} 2>&1
	@echo "###" >> ${OUTPUT}
	clang $(CFLAGS) example_flaws.c >> ${OUTPUT} 2>&1
	@echo "HTML scan result is under directory: ${OUTPUT_HTML}"
	scan-build -v -o ${OUTPUT_HTML} \
		-enable-checker alpha.core.BoolAssignment \
		-enable-checker alpha.core.CastSize \
		-enable-checker alpha.core.CastToStruct \
		-enable-checker alpha.core.FixedAddr \
		-enable-checker alpha.core.PointerArithm \
		-enable-checker alpha.core.PointerSub \
		-enable-checker alpha.core.SizeofPtr \
```

```
40                      -enable-checker alpha.deadcode.IdempotentOperations \
41                      -enable-checker alpha.deadcode.UnreachableCode \
42                      -enable-checker alpha.security.ArrayBoundV2 \
43                      -enable-checker alpha.security.MallocOverflow \
44                      -enable-checker alpha.security.ReturnPtrRange \
45                      -enable-checker alpha.unix.Chroot \
46                      -enable-checker alpha.unix.MallocWithAnnotations \
47                      -enable-checker alpha.unix.SimpleStream \
48                      -enable-checker alpha.unix.Stream \
49                      -enable-checker alpha.unix.cstring.BufferOverlap \
50                      -enable-checker alpha.unix.cstring.NotNullTerminated \
51                      -enable-checker alpha.unix.cstring.OutOfBounds \
52                      -enable-checker security.FloatLoopCounter \
53                      -enable-checker security.insecureAPI.rand \
54                      -enable-checker security.insecureAPI.strcpy \
55                      egcc -c example_flaws.c >> ${OUTPUT} 2>&1 # Replace egcc with
                            gcc in Linux
56  splint:
57          @echo "\n### $$(splint version 2>&1 |head -1) ###" >> ${OUTPUT} 2>&1
58          @echo "###" >> ${OUTPUT}
59          # ignore erroneous return value with '-'
60          -splint example_flaws.c >> ${OUTPUT} 2>&1
61  clean:
62          rm -rf *.o *.core codechecks.log ${OUTPUT_HTML}
```

D CODECHECKS.LOG

```
1  ### clang version 3.5 (trunk) and scan−build ###
2  ###
3  In file included from example_flaws.c:5:
4  /usr/include/stdio.h:396:23: warning: implicit conversion loses integer precision: 'int' to '
       unsigned char' [−Wconversion]
5                      return (∗_p−>_p++ = _c);
6                                       ~ ^~
7  example_flaws.c:18:24: warning: implicit conversion changes signedness: 'int' to 'unsigned
       long' [−Wsign−conversion]
8          unsigned long int a = −10;
9                                ~ ^~~
10 example_flaws.c:25:8: warning: comparison of integers of different signs: 'unsigned long' and
       'long' [−Wsign−compare]
11         if (a > b)
12             ~ ^ ~
13 example_flaws.c:49:17: warning: implicit conversion from 'long' to 'int' changes value from
       50159344557 to −1380262995 [−Wconstant−conversion]
14         signed int i = 0xbadbadbad;
15                      ~ ^~~~~~~~~~~
16 example_flaws.c:73:6: warning: implicit conversion loses integer precision: 'long' to 'int' [−
       Wshorten−64−to−32]
17         i = atol("1111111111111111");
18           ~ ^~~~~~~~~~~~~~~~~~~~~~~~
19 example_flaws.c:102:8: warning: comparing floating point with == or != is unsafe [−Wfloat
       −equal]
20         if (a == b)
21             ~ ^ ~
22 example_flaws.c:134:27: warning: implicit conversion changes signedness: 'int' to 'unsigned
       long' [−Wsign−conversion]
23         strncpy(myarr, mystring, len);
24         ~~~~~~~ ^~~
25 example_flaws.c:133:8: warning: implicit conversion loses integer precision: 'unsigned long' to
        'int' [−Wshorten−64−to−32]
26         len = strlen(mystring);
27             ~ ^~~~~~~~~~~~~~~~
28 example_flaws.c:151:11: warning: implicit conversion changes signedness: 'unsigned int' to '
       int' [−Wsign−conversion]
29         int i = 0xabad1dea;
30                      ~ ^~~~~~~~~~
31 9 warnings generated.
32
33 ### Cppcheck 1.67 ###
```

```
34  ###
35  Checking example_flaws.c...
36  [example_flaws.c:155]: (error) Width 32 given in format string (no. 1) is larger than
        destination buffer 'myarr[32]', use %31s to prevent overflowing it.
37  [example_flaws.c:171]: (error) Possible null pointer dereference: p
38  [example_flaws.c:185]: (error) Resource leak: fp
39  Checking example_flaws.c: __linux__...
40
41  ### flawdinfer 1.27 ###
42  ###
43  Flawfinder version 1.27, (C) 2001-2004 David A. Wheeler.
44  Number of dangerous functions in C/C++ ruleset: 160
45  Examining example_flaws.c
46  example_flaws.c:271: [4] (shell) system: This causes a new program to execute and is
        difficult to use safely. try using a library call that implements the same functionality if
        available.
47  example_flaws.c:284: [4] (shell) execv: This causes a new program to execute and is difficult
        to use safely. try using a library call that implements the same functionality if available.
48  example_flaws.c:212: [3] (misc) chroot: chroot can be very helpful, but is hard to use
        correctly. Make sure the program immediately chdir("/"), closes file descriptors, and
        drops root privileges, and that all necessary files (and no more!) are in the new root.
49  example_flaws.c:222: [3] (misc) chroot: chroot can be very helpful, but is hard to use
        correctly. Make sure the program immediately chdir("/"), closes file descriptors, and
        drops root privileges, and that all necessary files (and no more!) are in the new root.
50  example_flaws.c:73: [2] (integer) atol: Unless checked, the resulting number can exceed the
        expected range. If source untrusted, check both minimum and maximum, even if the
        input had no minus sign (large numbers can roll over into negative number; consider
        saving to an unsigned value if that is intended).
51  example_flaws.c:129: [2] (buffer) char: Statically-sized arrays can be overflowed. Perform
        bounds checking, use functions that limit length, or ensure that the size is larger than
        the maximum possible length.
52  example_flaws.c:141: [2] (buffer) char: Statically-sized arrays can be overflowed. Perform
        bounds checking, use functions that limit length, or ensure that the size is larger than
        the maximum possible length.
53  example_flaws.c:152: [2] (buffer) char: Statically-sized arrays can be overflowed. Perform
        bounds checking, use functions that limit length, or ensure that the size is larger than
        the maximum possible length.
54  example_flaws.c:163: [2] (buffer) char: Statically-sized arrays can be overflowed. Perform
        bounds checking, use functions that limit length, or ensure that the size is larger than
        the maximum possible length.
55  example_flaws.c:179: [2] (buffer) char: Statically-sized arrays can be overflowed. Perform
        bounds checking, use functions that limit length, or ensure that the size is larger than
        the maximum possible length.
```

```
56 example_flaws.c:182: [2] (misc) fopen: Check when opening files - can an attacker redirect
       it (via symlinks), force the opening of special file type (e.g., device files), move things
       around to create a race condition, control its ancestors, or change its contents?.
57 example_flaws.c:191: [2] (buffer) char: Statically-sized arrays can be overflowed. Perform
       bounds checking, use functions that limit length, or ensure that the size is larger than
       the maximum possible length.
58 example_flaws.c:194: [2] (misc) fopen: Check when opening files - can an attacker redirect
       it (via symlinks), force the opening of special file type (e.g., device files), move things
       around to create a race condition, control its ancestors, or change its contents?.
59 example_flaws.c:267: [2] (buffer) char: Statically-sized arrays can be overflowed. Perform
       bounds checking, use functions that limit length, or ensure that the size is larger than
       the maximum possible length.
60 example_flaws.c:133: [1] (buffer) strlen: Does not handle strings that are not \0-terminated
        (it could cause a crash if unprotected).
61 example_flaws.c:134: [1] (buffer) strncpy: Easily used incorrectly; doesn't always \0-
       terminate or check for invalid pointers.
62 example_flaws.c:155: [1] (buffer) scanf: it's unclear if the %s limit in the format string is
       small enough. Check that the limit is sufficiently small, or use a different input function.
63 example_flaws.c:270: [1] (port) snprintf: On some very old systems, snprintf is incorrectly
       implemented and permits buffer overflows; there are also incompatible standard
       definitions of it. Check it during installation, or use something else.
64
65 Hits = 18
66 Lines analyzed = 305 in 0.54 seconds (6809 lines/second)
67 Physical Source Lines of Code (SLOC) = 245
68 Hits@level = [0] 0 [1] 4 [2] 10 [3] 2 [4] 2 [5] 0
69 Hits@level+ = [0+] 18 [1+] 18 [2+] 14 [3+] 4 [4+] 2 [5+] 0
70 Hits/KSLOC@level+ = [0+] 73.4694 [1+] 73.4694 [2+] 57.1429 [3+] 16.3265 [4+] 8.16327
       [5+] 0
71 Minimum risk level = 1
72 Not every hit is necessarily a security vulnerability.
73 There may be other security vulnerabilities; review your code!
74
75 ### egcc (GCC) 4.8.3 ###
76 ###
77 example_flaws.c: In function 'example_signedness':
78 example_flaws.c:18:2: warning: negative integer implicitly converted to unsigned type [-
       Wsign-conversion]
79   unsigned long int a = -10;
80   ^
81 example_flaws.c:25:8: warning: comparison between signed and unsigned integer
       expressions [-Wsign-compare]
82   if (a > b)
83         ^
```

```
84  example_flaws.c: In function 'example_integeroverflow1':
85  example_flaws.c:49:2: warning: overflow in implicit constant conversion [-Woverflow]
86    signed int i = 0xbadbadbad;
87    ^
88  example_flaws.c: In function 'example_integeroverflow2':
89  example_flaws.c:73:10: warning: conversion to 'int' from 'long int' may alter its value [-
        Wconversion]
90    i = atol("1111111111111111");
91            ^
92  example_flaws.c: In function 'example_compare':
93  example_flaws.c:102:8: warning: comparing floating point with == or != is unsafe [-Wfloat
        -equal]
94    if (a == b)
95          ^
96  example_flaws.c: In function 'example_bufferoverflow':
97  example_flaws.c:133:14: warning: conversion to 'int' from 'size_t' may alter its value [-
        Wconversion]
98    len = strlen(mystring);
99                ^
100 example_flaws.c:134:2: warning: conversion to 'size_t' from 'int' may change the sign of the
         result [-Wsign-conversion]
101   strncpy(myarr, mystring, len);
102   ^
103 example_flaws.c: In function 'example_stacksmashing':
104 example_flaws.c:151:2: warning: conversion of unsigned constant value to negative integer
        [-Wsign-conversion]
105   int i = 0xabad1dea;
106   ^
107
108 ### Splint 3.1.2 --- 21 Nov 2014 ###
109 ###
110 Splint 3.1.2 --- 21 Nov 2014
111
112 example_flaws.c:7: Include file <unistd.h> matches the name of a POSIX library,
113     but the POSIX library is not being used. Consider using +posixlib or
114     +posixstrictlib to select the POSIX library, or -warnposix to suppress this
115     message.
116   Header name matches a POSIX header, but the POSIX library is not selected.
117   (Use -warnposixheaders to inhibit warning)
118 example_flaws.c: (in function example_signedness)
119 example_flaws.c:18:24: Variable a initialized to type int, expects unsigned
120                            long int: -10
121   To ignore signs in type comparisons use +ignoresigns
122 example_flaws.c:25:6: Operands of > have incompatible types (unsigned long int,
```

```
123                              long int): a > b
124  example_flaws.c: (in function example_integeroverflow2)
125  example_flaws.c:73:2: Assignment of long int to int:
126                              i = atol("1111111111111111")
127    To ignore type qualifiers in type comparisons use +ignorequals.
128  example_flaws.c: (in function example_integeroverflow2_fixed)
129  example_flaws.c:85:6: Unrecognized identifier: strtonum
130    Identifier used in code has not been declared. (Use -unrecog to inhibit
131    warning)
132  example_flaws.c:88:25: Duplicate long qualifier on non-int
133    Duplicate type qualifiers not supported by ISO standard. (Use -duplicatequals
134    to inhibit warning)
135  example_flaws.c:88:23: Format argument 1 to printf (%llu) expects unsigned int
136                               gets long long: i
137     example_flaws.c:88:17: Corresponding format code
138  example_flaws.c: (in function example_compare)
139  example_flaws.c:102:6: Dangerous equality comparison involving double types:
140                               a == b
141    Two real (float, double, or long double) values are compared directly using
142    == or != primitive. This may produce unexpected results since floating point
143    representations are inexact. Instead, compare the difference to FLT_EPSILON
144    or DBL_EPSILON. (Use -realcompare to inhibit warning)
145  example_flaws.c: (in function example_bufferoverflow)
146  example_flaws.c:133:2: Assignment of size_t to int: len = strlen(mystring)
147    To allow arbitrary integral types to match any integral type, use
148    +matchanyintegral.
149  example_flaws.c:134:27: Function strncpy expects arg 3 to be size_t gets int:
150                                len
151  example_flaws.c: (in function example_bufferoverflow_fixed)
152  example_flaws.c:144:2: Unrecognized identifier: strlcpy
153  example_flaws.c: (in function example_stacksmashing)
154  example_flaws.c:155:2: Return value (type int) ignored: scanf("%32s", myarr)
155    Result returned by function call is not used. If this is intended, can cast
156    result to (void) to eliminate message. (Use -retvalint to inhibit warning)
157  example_flaws.c:156:29: Format argument 2 to printf (%x) expects unsigned int
158                                gets int: i
159     example_flaws.c:156:16: Corresponding format code
160  example_flaws.c: (in function example_stacksmashing_fixed)
161  example_flaws.c:167:25: Function fgets expects arg 2 to be int gets size_t:
162                                sizeof((myarr))
163  example_flaws.c:171:3: Dereference of possibly null pointer p: *p
164    A possibly null pointer is dereferenced. Value is either the result of a
165    function which may return null (in which case, code should check it is not
166    null), or a global, parameter or structure field declared with the null
```

```
167    qualifier. (Use −nullderef to inhibit warning)
168     example_flaws.c:164:14: Storage p may become null
169  example_flaws.c:172:22: Possibly null storage myarr passed as non−null param:
170                                  printf (..., myarr)
171    A possibly null pointer is passed as a parameter corresponding to a formal
172    parameter with no /*@null@*/ annotation. If NULL may be used for this
173    parameter, add a /*@null@*/ annotation to the function parameter declaration.
174    (Use −nullpass to inhibit warning)
175  example_flaws.c: (in function example_nullcheck)
176  example_flaws.c:183:23: Function fgets expects arg 2 to be int gets size_t:
177                                  sizeof((linebuf))
178  example_flaws.c:183:34: Possibly null storage fp passed as non−null param:
179                                  fgets (..., fp)
180     example_flaws.c:182:7: Storage fp may become null
181  example_flaws.c:183:2: Return value (type char *) ignored: fgets(linebuf, s...
182    Result returned by function call is not used. If this is intended, can cast
183    result to (void) to eliminate message. (Use −retvalother to inhibit warning)
184  example_flaws.c: (in function example_nullcheck_fixed)
185  example_flaws.c:195:10: Null storage passed as non−null param: err (..., NULL)
186  example_flaws.c:196:27: Function fgets expects arg 2 to be int gets size_t:
187                                  sizeof((linebuf))
188  example_flaws.c:196:38: Possibly null storage fp passed as non−null param:
189                                  fgets (..., fp)
190     example_flaws.c:194:12: Storage fp may become null
191  example_flaws.c:203:3: Return value (type int) ignored: fclose(fp)
192  example_flaws.c: (in function example_chrootandprivdrop_fixed)
193  example_flaws.c:226:6: Operands of || are non−boolean (int):
194      setgroups(1, &newgid) || setresgid(newgid, newgid, newgid)
195    The operand of a boolean operator is not a boolean. Use +ptrnegate to allow !
196    to be used on pointers. (Use −boolops to inhibit warning)
197  example_flaws.c:228:6: Right operand of || is non−boolean (int):
198      setgroups(1, &newgid) || setresgid(newgid, newgid, newgid) ||
199      setresuid(newgid, newgid, newgid)
200  example_flaws.c: (in function example_omiterrno)
201  example_flaws.c:236:7: Unrecognized identifier: SIGPIPE
202  example_flaws.c: (in function example_inputvalidation)
203  example_flaws.c:270:2: Return value (type int) ignored: snprintf(cmd, si...
204  example_flaws.c:271:2: Return value (type int) ignored: system(cmd)
205  example_flaws.c: (in function example_inputvalidation_fixed)
206  example_flaws.c:278:24: Observer storage userinput used as initial value for
207                                  unqualified storage: cmd[1] = userinput
208    Observer storage is transferred to a non−observer reference. (Use
209    −observertrans to inhibit warning)
210  example_flaws.c:278:35: Local cmd[2] initialized to null value: cmd[2] = NULL
```

```
211     A reference with no null annotation is assigned or initialized to NULL. Use
212     /*@null@*/ to declare the reference as a possibly null pointer. (Use
213     -nullassign to inhibit warning)
214   example_flaws.c:284:3: Return value (type int) ignored: execv("/bin/ls",...
215   example_flaws.c: (in function main)
216   example_flaws.c:292:2: Return value (type [function (int) returns void])
217                              ignored: signal(SIGPIPE, ...
218   example_flaws.c:16:1: Function exported but not used outside example_flaws:
219                              example_signedness
220     A declaration is exported, but not used outside this module. Declaration can
221     use static qualifier. (Use -exportlocal to inhibit warning)
222      example_flaws.c:29:1: Definition of example_signedness
223   example_flaws.c:47:1: Function exported but not used outside example_flaws:
224                              example_integeroverflow1
225      example_flaws.c:54:1: Definition of example_integeroverflow1
226   example_flaws.c:67:1: Function exported but not used outside example_flaws:
227                              example_integeroverflow2
228      example_flaws.c:75:1: Definition of example_integeroverflow2
229   example_flaws.c:92:1: Function exported but not used outside example_flaws:
230                              example_compare
231      example_flaws.c:106:1: Definition of example_compare
232   example_flaws.c:126:1: Function exported but not used outside example_flaws:
233                               example_bufferoverflow
234      example_flaws.c:135:1: Definition of example_bufferoverflow
235   example_flaws.c:149:1: Function exported but not used outside example_flaws:
236                               example_stacksmashing
237      example_flaws.c:157:1: Definition of example_stacksmashing
238   example_flaws.c:176:1: Function exported but not used outside example_flaws:
239                               example_nullcheck
240      example_flaws.c:185:1: Definition of example_nullcheck
241   example_flaws.c:207:1: Function exported but not used outside example_flaws:
242                               example_chrootandprivdrop
243      example_flaws.c:214:1: Definition of example_chrootandprivdrop
244   example_flaws.c:233:1: Function exported but not used outside example_flaws:
245                               example_omiterrno
246      example_flaws.c:242:1: Definition of example_omiterrno
247   example_flaws.c:265:1: Function exported but not used outside example_flaws:
248                               example_inputvalidation
249      example_flaws.c:272:1: Definition of example_inputvalidation
250
251   Finished checking --- 42 code warnings
```

E Makefile of tvheadend (patch)

```
diff --git a/Makefile b/Makefile
index adfaa40..48c27f5 100644
--- a/Makefile
+++ b/Makefile
@@ -27,11 +27,23 @@ PROG := $(BUILDDIR)/tvheadend
 # Common compiler flags
 #

-CFLAGS += -Wall -Werror -Wwrite-strings -Wno-deprecated-declarations
+CC = clang
+CFLAGS += -Wall -Wwrite-strings
 CFLAGS += -Wmissing-prototypes -fms-extensions
 CFLAGS += -g -funsigned-char -O2
 CFLAGS += -D_FILE_OFFSET_BITS=64
+
+# <Own additions start>
+#CFLAGS += -pedantic
+#CFLAGS += -Wformat=2 -Wunused-parameter -Wsystem-headers \
+# -Wfloat-equal -Wshadow -Wpointer-arith -Wtype-limits \
+# -Wcast-qual -Wcast-align -Wconversion -Wempty-body \
+# -Wenum-compare -Wsign-compare -Wsizeof-pointer-memaccess \
+# -Waddress -Wredundant-decls -Wpointer-sign \
+# -Wstack-protector -fstack-protector -Woverlength-strings
+# </Own additions end>
 CFLAGS += -I${BUILDDIR} -I${ROOTDIR}/src -I${ROOTDIR}
+
 LDFLAGS += -ldl -lpthread -lm
 ifeq ($(CONFIG_LIBICONV),yes)
 LDFLAGS += -liconv
@@ -40,10 +52,12 @@ ifneq ($(PLATFORM), darwin)
 LDFLAGS += -lrt
 endif

+
 ifeq ($(COMPILER), clang)
-CFLAGS += -Wno-microsoft -Qunused-arguments -Wno-unused-function
-CFLAGS += -Wno-unused-value -Wno-tautological-constant-out-of-range-
    compare
-CFLAGS += -Wno-parentheses-equality -Wno-incompatible-pointer-types
+CFLAGS += -Wno-microsoft
+#CFLAGS += -Wno-microsoft -Qunused-arguments -Wno-unused-function
```

```
41 +#CFLAGS += -Wno-unused-value -Wno-tautological-constant-out-of-range-
       compare
42 +#CFLAGS += -Wno-parentheses-equality -Wno-incompatible-pointer-types
43  endif
44
45  vpath %.c $(ROOTDIR)
46 @@ -351,6 +365,16 @@ distclean: clean
47         rm -rf ${ROOTDIR}/build.*
48         rm -f ${ROOTDIR}/.config.mk
49
50 +cppcheck:
51 +  cppcheck -q -f -I${BUILDDIR} -I${ROOTDIR}/src -I${ROOTDIR} src >cppcheck.
       txt 2>&1
52 +
53 +flawfinder:
54 +  flawfinder --html --context src/ >flawfinder.html
55 +
56 +splint:
57 +  rm -f splintoutput.txt
58 +  ./splint.sh
59 +
60  # Create version
61  $(BUILDDIR)/src/version.o: $(ROOTDIR)/src/version.c
62  $(ROOTDIR)/src/version.c: FORCE
```

F splint.sh

```
1 #!/bin/sh
2
3 for line in $(find src/ -type f -iname "*.c"); do
4         splint -warnposix "$line" >> splintoutput.txt 2>&1
5 done
```