



EÖTVÖS LORÁND UNIVERSITY

FACULTY OF INFORMATICS

DEPARTMENT OF PROGRAMMING LANGUAGES  
AND COMPILERS

# Finding Security Vulnerabilities with Static Code Analysis

*Supervisor:*

Porkoláb Zoltán

Associate Professor, PhD

*Author:*

Tsinadze Zurab

Computer Science BSc

*Budapest, 2021*

## Thesis Registration Form

**Student's Data:**

**Student's Name:** Tsinadze Zurab

**Student's Neptun code:** PRRXNH

**Course Data:**

**Student's Major:** Computer Science BSc

I have an internal supervisor

**Internal Supervisor's Name:** Porkoláb Zoltan

Supervisor's Home Institution: ELTE Faculty of Informatics, Dept. of Programming Languages and Compilers

Address of Supervisor's Home Institution: H-1117 Pázmány Péter stny. 1/c Budapest

Supervisor's Position and Degree: Associate Professor, PhD.

**Thesis Title:** Finding Security Vulnerabilities with Static Code Analysis

**Topic of the Thesis:**

*(Upon consulting with your supervisor, give a 150-300-word-long synopsis of your planned thesis. )*

Static Code Analyzers perform analysis on computer software without actually executing it. Warnings emitted by such tools gives developer free, fast, and anonymous code review. Static Analysis can be used to check for certain types of problems, for which it is impossible to write tests, for example, mistakes that cause undefined behavior. LLVM's open-source Clang Static Analyzer is one of the most popular and powerful tools for C, C++, and Objective-C code. The SEI CERT C/C++ Coding Standard is a set of rules and recommendations for secure coding. Failing to enforce any of these rules could result in a serious security problem and could cost companies their reputation, as well as money, and face legal issues. Hence, it would be good if enforcing such rules could be automated. One way to do this is by using Static Code Analyzers. The aim of the thesis work will be to extend the static analyzer toolset of the LLVM framework to cover SEI CERT Rules related to incorrect usage of the Environment. The program will be written in C++. The Output of the work is planned to be open-sourced as part of the LLVM framework.

Budapest, 2020.11.24.

# Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	Motivation . . . . .	3
1.2	CERT C/C++ Coding Standard Rules . . . . .	4
1.2.1	POS34-C . . . . .	4
1.2.2	ENV31-C . . . . .	5
1.2.3	ENV34-C . . . . .	6
1.3	Static Analysis . . . . .	6
1.3.1	Static Analysis Techniques . . . . .	7
1.3.2	Clang Static Analyzer . . . . .	9
<b>2</b>	<b>User Documentation</b>	<b>11</b>
2.1	Install Guide . . . . .	11
2.1.1	System Requirements . . . . .	11
2.1.2	Building from source . . . . .	12
2.2	Running Analysis . . . . .	13
2.3	Output options . . . . .	14
2.3.1	Text output . . . . .	15
2.3.2	HTML output . . . . .	15
2.3.3	scan-build . . . . .	16
2.3.4	CodeChecker . . . . .	17
<b>3</b>	<b>Developer documentation</b>	<b>21</b>
3.1	Prerequisites . . . . .	21
3.2	Building Clang . . . . .	21
3.3	Clang Static Analyzer Basics . . . . .	23
3.3.1	Exploded Graph . . . . .	23

3.3.2	Checkers . . . . .	24
3.3.3	Custom Program States . . . . .	25
3.4	Implementation . . . . .	25
3.4.1	alpha.security.cert.pos.34c . . . . .	25
3.4.2	alpha.security.cert.env.InvalidPtr . . . . .	29
3.5	Testing . . . . .	33
3.5.1	Lit . . . . .	33
3.5.2	Running analysis on large code bases . . . . .	40
<b>4</b>	<b>Conclusion and future work</b>	<b>43</b>
<b>A</b>	<b>List of created and modified files</b>	<b>45</b>
A.1	List of modified files for POS34 . . . . .	45
A.2	List of added files for POS34 . . . . .	45
A.3	List of modified files for InvalidPtr . . . . .	45
A.4	List of added files for InvalidPtr . . . . .	46
	<b>Bibliography</b>	<b>47</b>
	<b>List of Figures</b>	<b>50</b>
	<b>List of Tables</b>	<b>51</b>

# Chapter 1

## Introduction

### 1.1 Motivation

Bugs are more common in programming languages with access to low-level system resources such as C and C++. A variety of scenarios can result in so-called an undefined or implementation-dependent behavior. Many examples of such problematic code are hidden in massive language standard specifications and are easily overlooked by inexperienced developers. Not to mention that even well-known bugs (division by zero, use of an uninitialized variable, use after free, etc.) are difficult to detect when they occur not in a single point of the code, but rather in a long chain of events that lead up to it.

The real issue with undefined behavior (UB) is that a program may behave correctly on some runs but produce runtime errors on others, depending on the compiler, platform, build configurations, and so on.

In the best-case scenario, UB can cause minor, sometimes even tolerable problems, but it can also be the source of irreparable damage; for example, a crash during an important execution can result in tragedy or severe financial loss. Patriot Missile Error [1], Knight's 440 Million Error [2], Heathrow Terminal 5 Opening [3], Ariane 5 Flight 501 [4], and many more are well-known examples. Undefined behavior can allow black hat hackers to gain access to confidential information (Hearthbleed [5]) or run malicious code on the target computer.

Even if it is difficult, it is still the developer's responsibility to avoid such errors. Static analyzers, which are automated tools for finding bugs, come to the rescue.

The goal of this thesis is to extend the open-source static analysis tool so that it can detect three more undefined or implementation-dependent behavior scenarios.

## 1.2 CERT C/C++ Coding Standard Rules

The Carnegie Mellon Software Engineering Institute and thousands of researchers and language experts collaborated to create the Secure Coding standard, which describes the root causes of common software vulnerabilities [6] [7]. The following SEI CERT Rules will be addressed in the thesis: POS34-C [8], ENV31-C [9], and ENV34-C [10].

### 1.2.1 POS34-C

**Do not call `putenv()` with a pointer to an automatic variable as the argument**

POSIX [11] function *putenv* is used to change or add environment variables. It accepts only one argument, a pointer to a string of the form "name=value".

*putenv* does not make a copy of the passed string; instead, it adds the pointer to the environment array directly. When a pointer to an automatic variable is passed as an argument, a garbage value may end up in the environment because the containing function's stack memory is recycled [12]. This can result in unexpected program behavior or even give the attacker the ability to run arbitrary code.

```
1 int volatile_memory_example(const char *var) {  
2     char env[1024];  
3     int retval = snprintf(env, sizeof(env), "TEST=%s", var);  
4     if (retval < 0 || (size_t)retval >= sizeof(env)) {  
5         /* Handle error */  
6     }  
7     return putenv(env);  
8 }
```

Code 1.1: Pointer to automatic buffer passed to `putenv()`. Example from CERT rule page.

To avoid the issue, programmer has few options:

- Add `static` keyword on line 2.
- Use Heap memory allocated pointer as an argument to `putenv()`.
- Instead of `putenv()` favor `setenv()` function, which allocates heap memory for environment variables.

### 1.2.2 ENV31-C

**Do not rely on an environment pointer following an operation that may invalidate it**

In ISO C standard `main` function takes no arguments, or takes two - `int argc` and `char *argv[]`; however, in Unix systems, a third argument, `char *envp[]`, can be used, which points to the program's environment and has the same value as `environ` (global array of environment variables [13]) [14].

Any change to the environment, such as calling `putenv`, `setenv`, or their variants, may result in memory being reallocated. `environ` has been updated to reflect this change, but `envp` has not, so its use may result in unexpected behavior.

```
1 int main(int argc, const char *argv[], const char *envp[]) {  
2     putenv((char *) "VARIABLE=VALUE"); // envp invalidated  
3  
4     if (envp != NULL) {  
5         for (int i = 0; envp[i] != NULL; ++i) {  
6             puts(envp[i]);  
7         }  
8     }  
9     return 0;  
10 }
```

Code 1.2: `envp` pointer used after the modification of environment.

Developers should keep in mind that changes to the environment will not be reflected in `envp` and as an alternative using `environ` (or `_environ` on Windows standard [15]) is an option.

### 1.2.3 ENV34-C

#### Do not store pointers returned by certain functions

Some functions copy their results into the internal static buffer and return a pointer to it, but the buffer is overwritten for each subsequent call. As a result, the previous return value should not be used any longer.

These functions include *getenv*, *asctime*, *localeconv*, *setlocale* and *strerror*.

```
1 void invalidated_pointer_usage(){  
2     char *p, pp;  
3     p = getenv("VAR1");  
4     pp = getenv("VAR2"); // p invalidated  
5     *p; // dereference of invalid pointer  
6 }
```

Code 1.3: Invalidated pointer usage.

A copy of the first pointer should be created if the programmer wants to use it after the subsequent call. Another solution would be to use Annex K implementations of these functions, but Annex K is still being discussed, and these functions could ultimately be withdrawn [16].

## 1.3 Static Analysis

Static Analysis tools do not run the program, but rather inspect the source or generated binary code at compile time. It is a broad topic that includes code verification, code transformation, optimization techniques, etc. However, in this work, we consider static analyzers to be bug-finding tools.

Static analysis is a difficult, if not impossible, problem. We can, however, approximate it by employing some heuristics. This, of course, comes at a cost in the form of False Positives (false reports) and False Negatives (missing some real problems). Any analyzer tool's goal is to reduce these two as much as possible.

In this section, we'll go over static analysis techniques and reason why we chose Clang Static Analyzer [17] for this project.



### 1.3.1 Static Analysis Techniques

#### Textual Pattern Matching

The concept behind this technique is straightforward. First, the code is converted to canonical form, and then we match different patterns to it. Pattern matching can be used to find relatively simpler errors, for example, if we literally have `" / 0 "` in our code. It is very fast and relatively easy to implement. However, such a crude approach will not serve our purpose.

#### AST Matching

This method attempts to match the abstract syntax tree. AST is an intermediate representation of the code. When compared to textual form, we have much more information available thanks to the compiler, for example, types, templates are resolved, etc. The Abstract Syntax Tree of a simple code snippet is shown in figure 1.1.

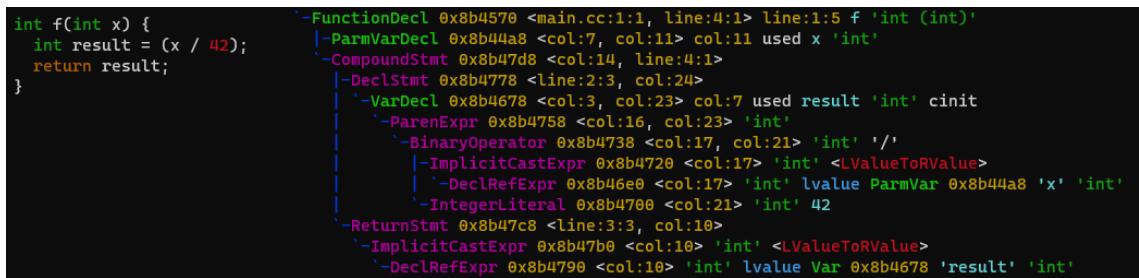


Figure 1.1: Clang AST example

Although AST matching is clearly superior to Textual Pattern Matching, snippet 1.4 demonstrates that POS34 is a control flow dependent problem that cannot be solved using this technique. Similar arguments can be made for the other two rules.

```

1 void foo() {
2     char *buffer = "X=Y";
3     if (rand() % 2 == 0) {
4         buffer = malloc(4);
5         strcpy(buffer, "X=Y");
6     }
7     putenv(buffer); // is buffer on heap or stack?
8 }

```

```

9 void bar() {
10     char *buffer = malloc(4);
11     strcpy(buffer, "X=Y");
12     if (rand() % 2 == 0) {
13         free(buffer);
14         buffer = "X=Y";
15     }
16     putenv(buffer); // is buffer on heap or stack?
17 }

```

Code 1.4: POS34 depends on control flow

## Symbolic Execution

The Symbolic Execution technique simulates program execution, tries to enumerate every possible execution path, and assigns symbols to represent unknown values. It stores the constraints on symbols (see figure 1.2) and eliminates unreachable paths. Such an execution comes at a high cost (exponential in the worst case), but it produces very precise results and allows for the detection of more complex bugs.

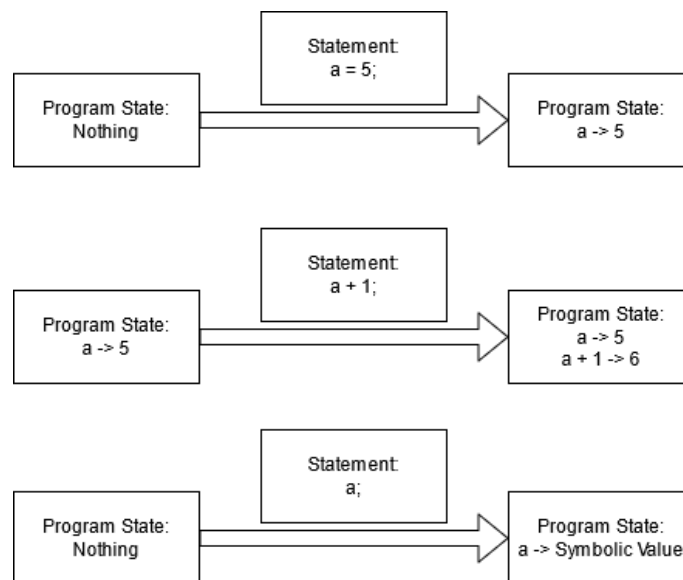


Figure 1.2: How statements affect the program state

### 1.3.2 Clang Static Analyzer

Clang Static Analyzer is a sophisticated, modern automated tool for detecting flaws in C, C++, and Objective-C codebases. It is included in the Clang compiler and makes use of all of LLVM's algorithms and data structures.

The tool is developed and used by technology giants like Ericsson, Apple, Google, Microsoft, Mozilla, and others together with the open-source community and academia. Clang SA is a path-sensitive analyzer that also performs symbolic execution, has excellent memory modeling [18], and is context-sensitive. All of this enables Clang Static Analyzer to detect more sophisticated problems, such as bad sequences of events in your program.

Finding the bug is only part of the analyzer's job; equally important is how the tool communicates the report to the developer, especially if it is the result of a long chain of events. Clang SA displays the full path, how the problem could have occurred. Figure 1.3 shows analysis result for the Code 1.5.

Furthermore, the tool is extensible by implementing new modules known as checkers [19], allowing us to detect new types of bugs.

All of these factors combine to make Clang Static Analyzer an ideal tool for this project.

```
1 int foo(int x) {  
2     int y = x;  
3     if (y == 0) {  
4         return 1/x;  
5     }  
6     return 0;  
7 }
```

Code 1.5: Invalidated pointer usage.

```
int foo(int x) {  
    int y = x;  
    if (y == 0) {  
        1 Assuming 'y' is equal to 0 →  
        2 ← Taking true branch →  
        return 1/x;  
        3 ← Division by zero  
    }  
    return 0;  
}
```

Figure 1.3: Static Analyzer warning example

# Chapter 2

## User Documentation

In this chapter, the Clang Static Analyzer will be explained from an end-user perspective. Clang SA is a Clang compiler library that can be found in the LLVM project repository [20].

This work implements two new checkers, *alpha.security.cert.pos.34c* for POS34-C and *alpha.security.cert.env.InvalidPtr* to cover both ENV31-C and ENV34-C. At the time of writing, the first checker is already included in LLVM, while the second is still undergoing official review.

### 2.1 Install Guide

#### 2.1.1 System Requirements

Table 2.1 shows the system requirements and supported compilers for building LLVM. The checkers were developed with Ubuntu 20.04 and tested on Ubuntu 18.04, macOS, and WSL Ubuntu 20.04.

Operating System	Processor Architecture	Compiler
Linux	x861	gcc, clang
Linux	amd64	gcc, clang
Linux	arm	gcc, clang
Linux	Mips	gcc, clang
Linux	PowerPC	gcc, clang
Solaris	V9	gcc
FreeBSD	x861	gcc, clang
FreeBSD	amd64	gcc, clang
NetBSD	x861	gcc, clang
NetBSD	amd64	gcc, clang
macOS2	PowerPC	gcc
macOS	x86	gcc, clang
Cygwin	x86	gcc
Windows	x86	Visual Studio
Windows64	x86-64	Visual Studio

Table 2.1: Clang Static Analyzer system requirements

### 2.1.2 Building from source

The commands below will compile LLVM from source. Note that prerequisites are *git*, *CMake* (version 3.4.3 or higher), *gcc* (version 5.1.0 or higher), and *Ninja* build system [21].

```
1 # on Windows --config core.autocrlf=false flag need to be added to
   the git clone command.
2 git clone https://github.com/llvm/llvm-project.git
3 cd llvm-project
4
5 # POS34 checker is already part of LLVM, however for ENV checker
   user needs to apply git patch
6 # on top of c79bc5942d0efd4740c7a6d36ad951c59ef3bc0e
```

```
7 # Author: Stefan Pintilie <stefanp@ca.ibm.com>, Date: Tue May 11
   05:32:32 2021 -0500
8 # It could work with the current HEAD of llvm-project, but recent
   changes might have caused merge conflict.
9
10 git checkout c79bc5942d0efd4740c7a6d36ad951c59ef3bc0e
11 git apply ../invalidPtrChecker.diff
12
13 mkdir build
14 cd build
15
16 # Configure build using cmake
17 cmake \
18     -G "Ninja" \
19     -DLLVM_ENABLE_PROJECTS="llvm;clang;clang-tools-extra" \
20     -DCMAKE_BUILD_TYPE=Release \
21     -DBUILD_SHARED_LIBS=ON \
22     -DLLVM_TARGETS_TO_BUILD=X86 \
23     -DCMAKE_EXPORT_COMPILE_COMMANDS=ON \
24     ../llvm/
25
26 # -j 4 is number of parallel jobs, user can set it to more.
27 # It will take a while...
28 ninja -j 4
29
30 # For ease of use the user can add built binaries to the PATH
31 # export PATH="path/to/llvm-project/build/bin:$PATH"
```

## 2.2 Running Analysis

To demonstrate how to use Clang Static Analyzer, we create main.c file with the following code:

```
1 #include <stdlib.h>
2 #include <stdio.h>
3
4 int call_putenv(const char *var) {
5     char env[1024];
```

```
6  int retval = snprintf(env , sizeof(env), "TEST=%s", var);
7  if (retval < 0) {
8      // handle error
9  }
10 return putenv(env); // putenv function should not be called with
    auto variables
11 }
12
13 int main(){
14     call_putenv("hello clang!");
15 }
```

The simplest way to run Clang SA is to use the clang compiler with the `--analyze` option. In order to activate our checkers, we must also include `-analyzer-checker=<checker name>` flag.

```
1 clang main.c --analyze \
2 -Xclang -analyzer-checker=alpha.security.cert.pos.34c
```

If the installation was successful, running the above command gives the following output:

```
zuka@cc:~$ clang main.c --analyze -Xclang -analyzer-checker=alpha.security.cert.pos.34c
main.c:10:10: warning: The 'putenv' function should not be called with arguments that have automatic storage [alpha.security.cert.pos.34c]
    return putenv(env); // putenv function should not be called with auto variables
           ^
1 warning generated.
```

Figure 2.1: POS34-C checker output

Similarly to the POS34-C, the following command can be used to check for ENV31 and ENV34 violations:

```
1 clang main.c --analyze \
2 -Xclang -analyzer-checker=alpha.security.cert.env.InvalidPtr
```

## 2.3 Output options

More analyzer-specific flags can be added to the invocation commands seen in 2.2. Users, for example, can choose the best output format for their needs:

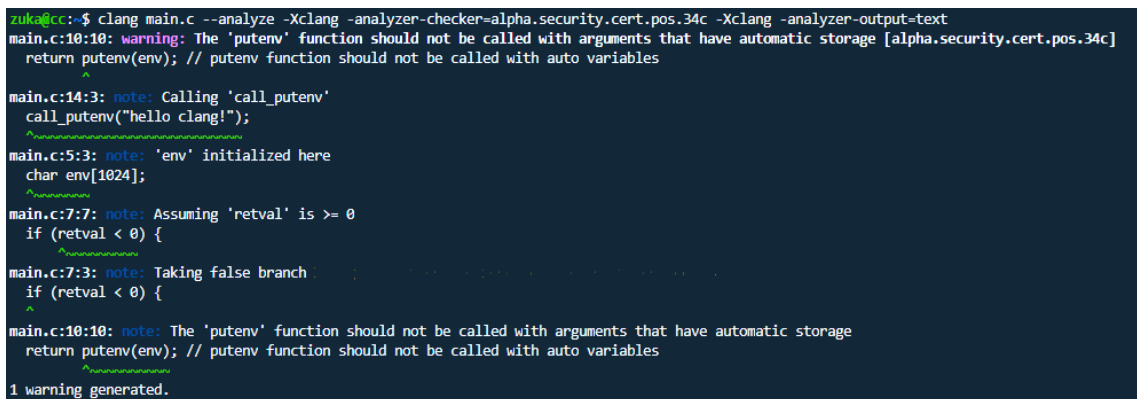
```
1 clang main.c --analyze \
2 -Xclang -analyzer-checker=alpha.security.cert.env.InvalidPtr \
3 -Xclang -analyzer-output=<output type>
```



Running the command without the `analyzer-output` flag results in default value "none", i.e. it gives warning in standard output as shown in figure 2.1.

### 2.3.1 Text output

`-analyzer-output=text` still displays the analysis result in the terminal, but it also shows the path diagnostic messages. Following the actual warning, we can see the steps taken by the symbolic execution that resulted in the bug.



```
zuka@cc:~$ clang main.c --analyze -Xclang -analyzer-checker=alpha.security.cert.pos.34c -Xclang -analyzer-output=text
main.c:10:10: warning: The 'putenv' function should not be called with arguments that have automatic storage [alpha.security.cert.pos.34c]
    return putenv(env); // putenv function should not be called with auto variables
           ^
main.c:14:3: note: Calling 'call_putenv'
    call_putenv("hello clang!");
    ^
main.c:5:3: note: 'env' initialized here
    char env[1024];
    ^
main.c:7:7: note: Assuming 'retval' is >= 0
    if (retval < 0) {
    ^
main.c:7:3: note: Taking false branch
    if (retval < 0) {
    ^
main.c:10:10: note: The 'putenv' function should not be called with arguments that have automatic storage
    return putenv(env); // putenv function should not be called with auto variables
           ^
1 warning generated.
```

Figure 2.2: POS34-C with diagnostics

### 2.3.2 HTML output

The analysis results can also be viewed in HTML format, with a simple graphical interface.

```
1 clang main.c --analyze \  
2 -Xclang -analyzer-checker=alpha.security.cert.env.InvalidPtr  
3 -Xclang -analyzer-output=html  
4 -o output/
```

If we run the above command will create a new "output" directory (if one does not already exist) where the analysis results will be stored. In our case, it will generate a single (depending on the number of warnings in our source code) HTML file, which will look like this when viewed in a browser:

**Bug Summary**

File: /home/zuka/main.c

Warning: [line 10, column 10](#)

The 'putenv' function should not be called with arguments that have automatic storage

**Annotated Source Code**Press [?](#) to see keyboard shortcuts[Show analyzer invocation](#)☐ Show only relevant lines

```
1  #include <stdlib.h>
2  #include <stdio.h>
3
4  int call_putenv(const char *var) {
5      char env[1024];
6
7      int retval = snprintf(env, sizeof(env), "TEST=%s", var);
8      if (retval < 0) {
9          // handle error
10     }
11     return putenv(env); // putenv function should not be called with auto variables
12 }
13
14 int main(){
15     call_putenv("hello clang!");
16 }
```

2 ← 'env' initialized here →

3 ← Assuming 'retval' is >= 0 →

4 ← Taking false branch →

5 ← The 'putenv' function should not be called with arguments that have automatic storage

1 ← Calling 'call\_putenv' →

Figure 2.3: HTML output example

### 2.3.3 scan-build

scan-build [22] is a command-line utility that acts as a wrapper around analysis invocation.

The command's general format is as follows:

```
scan-build [scan-build options] <command> [command options]
```

It generates HTML reports and displays warnings in the terminal.

When `-o` is not used to specify the output directory, it creates a temp folder with the current timestamp as the name by default.

Running "scan-view" suggested at the last line of output will start webserver where analysis can be examined.

To browse the list of found bugs, a single `index.html` file is generated, and by opening a specific report, we see a page similar to figure 2.3.

```
zuka@cc:~$ scan-build -enable-checker alpha.security.cert.pos.34c -o . clang main.c
scan-build: Using '/home/zuka/llvm-project/build/bin/clang-12' for static analysis
main.c:10:10: warning: The 'putenv' function should not be called with arguments that have automatic storage [alpha.security.cert.pos.34c]
    return putenv(env); // putenv function should not be called with auto variables
           ^~~~~~
1 warning generated.
scan-build: Analysis run complete.
scan-build: 1 bug found.
scan-build: Run 'scan-view /home/zuka/2021-03-28-034050-23009-1' to examine bug reports.
```

Figure 2.4: scan-build output

The main benefit of scan-build is that the second part of its invocation can contain any command, allowing users to run static analysis as part of performing a regular build, for example:

```
scan-build -enable-checker alpha.security.cert.env.InvalidPtr -o . make -j4
```

Moreover, `--use-analyzer=<path to clang>` flag can also be added, if the user wishes to use the specific clang build version.

### 2.3.4 CodeChecker

Ericsson developed the CodeChecker [23] tool in collaboration with Eötvös Loránd University to replace the scan-view utility. In this section, we will show you how to use CodeChecker to analyze an open-source "git" project.

It should be noted that CodeChecker is only available for Linux and Mac OS.

#### Setting up CodeChecker

To build the CodeChecker from source files, use the following commands:

```
1 # prerequisites are npm, virtualenv
2
3 git clone https://github.com/Ericsson/CodeChecker.git --depth 1 ~/
   codechecker
4 cd ~/codechecker
5
6 # create virtual environment and activate it
7 make venv # or venv_osx for mac
8 source $PWD/venv/bin/activate
9
```

```
10 # build codechecker
11 make package
12
13 # optionally user can add it to path for ease of use:
14 # export PATH="$PWD/build/CodeChecker/bin:$PATH"
15
16 cd ..
```

## Configuring clang version

CodeChecker detects and uses the latest available version of Clang that is in the user's path. Since we wish to use the custom-built clang for analysis, we must specify it inside configuration file `~/codechecker/build/CodeChecker/config/package_layout.json` Value of "clangsa" should be set to `/path/to/llvm-project/build/bin/clang`.

## Running analysis on git

```
1 # install prerequisites for building git
2 sudo apt-get install gettext
3 git clone https://github.com/git/git.git
4 cd git
5 make configure
6 ./configure --prefix=/usr
7 CodeChecker log -b "make all -j42" -o compile_commands.json
```

The commands above will generate a JSON file containing a compilation database for the git build process. This file is passed to the `CodeChecker analyze`, which performs the analysis on each compile command.

```
1 CodeChecker analyze \
2   /path/to/compile_commands.json \
3   --analyzers clangsa \
4   --enable alpha.security.cert.env.InvalidPtr \
5   -o /path/to/output/dir/ \
6   -j 42
```

This will generate analysis results in `.plist` files, which can then be converted to HTML with the `CodeChecker parse` command.



Figure 2.5: CodeChecker html output

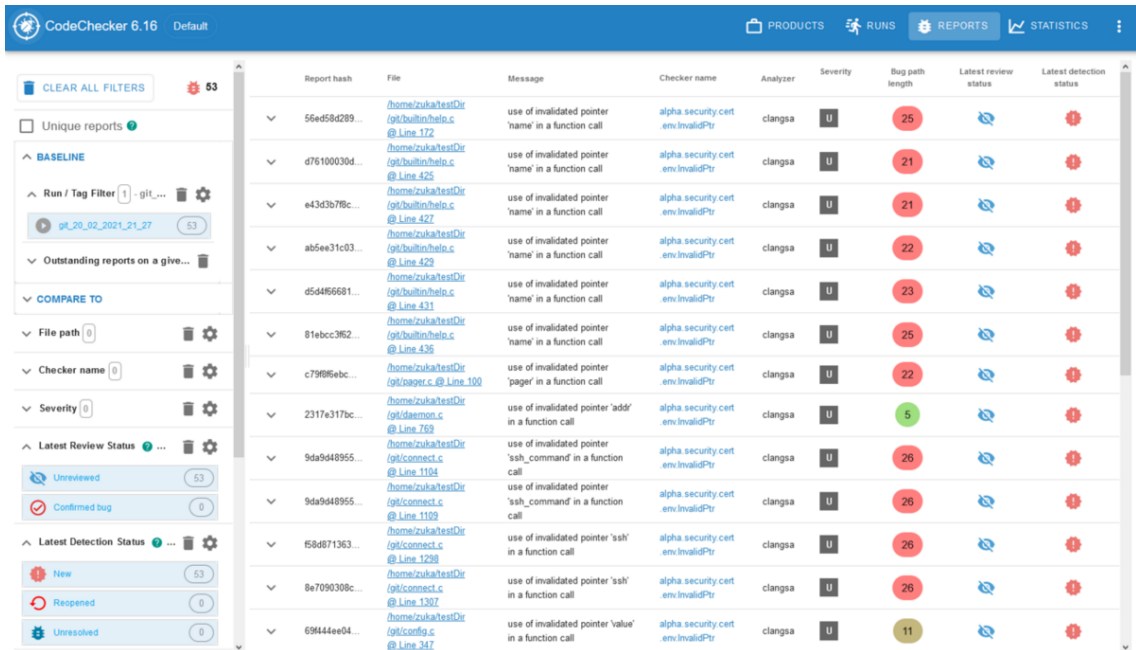
CodeChecker allows users to save analysis results on its server, providing them with a sophisticated graphical interface.

```

1 # start the server on localhost:8555/
2 CodeChecker server --workspace ./ws --port 8555
3
4 # store the analysis results in CC database
5 CodeChecker store \
6   /path/to/plist/files/ \
7   --name "project name" \
8   --url http://localhost:8555/Default

```

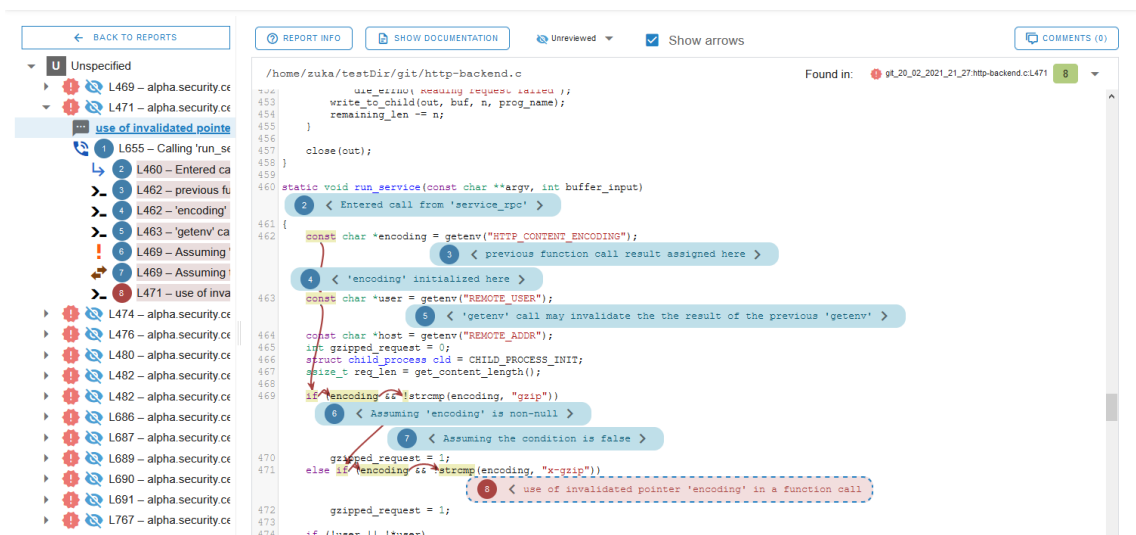
## 2. User Documentation



The screenshot shows the CodeChecker 6.16 interface. On the left, there's a sidebar with filters: 'Unique reports' (1), 'BASELINE' (Run / Tag Filter: 1 - git..., 53), 'COMPARE TO' (File path, Checker name, Severity), 'Latest Review Status' (Unreviewed: 53, Confirmed bug: 0), and 'Latest Detection Status' (New: 53, Reopened: 0, Unresolved: 0). The main area displays a table of reports.

Report hash	File	Message	Checker name	Analyzer	Severity	Bug path length	Latest review status	Latest detection status
56ed58d289...	/home/zuka/testDir/ /git/builtins/hello.c @ Line 112	use of invalidated pointer 'name' in a function call	alpha.security.cert .env.InvalidPtr	clangsa	U	25		
d7610030d...	/home/zuka/testDir/ /git/builtins/hello.c @ Line 425	use of invalidated pointer 'name' in a function call	alpha.security.cert .env.InvalidPtr	clangsa	U	21		
e43d3b78c...	/home/zuka/testDir/ /git/builtins/hello.c @ Line 427	use of invalidated pointer 'name' in a function call	alpha.security.cert .env.InvalidPtr	clangsa	U	21		
ab5ee31c03...	/home/zuka/testDir/ /git/builtins/hello.c @ Line 429	use of invalidated pointer 'name' in a function call	alpha.security.cert .env.InvalidPtr	clangsa	U	22		
d5d486681...	/home/zuka/testDir/ /git/builtins/hello.c @ Line 431	use of invalidated pointer 'name' in a function call	alpha.security.cert .env.InvalidPtr	clangsa	U	23		
81ebcc362...	/home/zuka/testDir/ /git/builtins/hello.c @ Line 436	use of invalidated pointer 'name' in a function call	alpha.security.cert .env.InvalidPtr	clangsa	U	25		
c79896bec...	/home/zuka/testDir/ /git/pager.c @ Line 100	use of invalidated pointer 'pager' in a function call	alpha.security.cert .env.InvalidPtr	clangsa	U	22		
2317e317bc...	/home/zuka/testDir/ /git/daemon.c @ Line 759	use of invalidated pointer 'addr' in a function call	alpha.security.cert .env.InvalidPtr	clangsa	U	5		
9da948955...	/home/zuka/testDir/ /git/connect.c @ Line 1104	use of invalidated pointer 'ssh_command' in a function call	alpha.security.cert .env.InvalidPtr	clangsa	U	26		
9da948955...	/home/zuka/testDir/ /git/connect.c @ Line 1109	use of invalidated pointer 'ssh_command' in a function call	alpha.security.cert .env.InvalidPtr	clangsa	U	26		
58d871363...	/home/zuka/testDir/ /git/connect.c @ Line 1298	use of invalidated pointer 'ssh' in a function call	alpha.security.cert .env.InvalidPtr	clangsa	U	26		
8e709030c...	/home/zuka/testDir/ /git/connect.c @ Line 1307	use of invalidated pointer 'ssh' in a function call	alpha.security.cert .env.InvalidPtr	clangsa	U	26		
69644ee04...	/home/zuka/testDir/ /git/config.c @ Line 347	use of invalidated pointer 'value' in a function call	alpha.security.cert .env.InvalidPtr	clangsa	U	11		

Figure 2.6: CodeChecker graphical user interface, list of reports



The screenshot shows the 'REPORT INFO' view for report L471. The left sidebar lists various reports, with 'use of invalidated pointer' selected. The main area displays the source code for `/home/zuka/testDir/git/http-backend.c` with annotations and arrows indicating the path of the bug. The annotations include: 'Entered call from 'service\_rpc' >', '< previous function call result assigned here >', '< 'encoding' initialized here >', '< 'getenv' call may invalidate the the result of the previous 'getenv' >', '< Assuming 'encoding' is non-null >', and '< Assuming the condition is false >'. The bug is identified as 'use of invalidated pointer 'encoding' in a function call'.

Figure 2.7: CodeChecker graphical user interface, path diagnostics

# Chapter 3

## Developer documentation

This chapter will cover the basics and the process of developing the Clang Static Analyzer checker. The developer is presumed to have the advanced knowledge of C++, the user experience of Linux, and familiarity with some of the build systems.

### 3.1 Prerequisites

Before writing the code, the developer must have a few necessary tools.

- LLVM repository, available from GitHub [20]
- Git version control system [24]
- CMake
- Ninja build system. Other build systems can be used as well, however most LLVM developers use ninja
- Development environment of your choice. VS Code, vim, CLion are viable options among others

### 3.2 Building Clang

The first step is to checkout LLVM repository from GitHub and create *build* directory inside. CMake is used to generate build files, it takes few parameters,

which are self-explanatory. We use Ninja to start the build process and lastly, it is always a good idea to run all the tests before we start the development process.

```
1 git clone https://github.com/llvm/llvm-project.git
2
3 cd llvm-project
4 mkdir build
5 cd build
6
7 # Configure build using cmake
8 cmake \
9     -G "Ninja" \
10     -DLLVM_ENABLE_PROJECTS="llvm;clang;clang-tools-extra" \
11     -DCMAKE_BUILD_TYPE=RelWithDebInfo \
12     -DBUILD_SHARED_LIBS=ON \
13     -DLLVM_TARGETS_TO_BUILD=X86 \
14     -DCMAKE_EXPORT_COMPILE_COMMANDS=ON \
15     -DLLVM_ENABLE_ASSERTIONS=ON \
16     ../llvm/
17
18 # -j 4 is number of parallel jobs
19 # It will take a while...
20 ninja -j 4
21
22 # Run all the tests
23 ninja check-all -j 4
24
25 # For ease of use user can add built binaries to the PATH
26 # export PATH="path/to/llvm-project/build/bin:$PATH"
```

The above steps will build clang along with some useful tools for development and testing. Later on, it is enough to rebuild just clang using `ninja clang -j 4` while running only clang analysis specific tests is possible with `ninja check-clang-analysis`



## 3.3 Clang Static Analyzer Basics

### 3.3.1 Exploded Graph

Static Analysis starts with reading plain source code, which is later transformed to Abstract Syntax Tree by Clang. Clang-Tidy, another LLVM native static analyzer tool, is using AST to find bugs in a source code, however, Clang SA takes a different approach and dives into Control Flow Graph. Clang's CFG is a data structure that consists of AST statements, which are ordered as they are actually executed (figure 3.1). CFG is used by Clang to produce some compiler warnings, but Static Analyzer goes one step further and produces a new data structure called Exploded Graph. It essentially represents the result of the analysis, hence Clang Static Analyzer can be seen as a converter from AST to the Exploded Graph. Finding a bug in the source code is reduced to solving the graph reachability problem.

Exploded Graph consists of paths through CFG. Nodes are pairs of Program Point (the point between two statements, i.e. we finished the first but haven't started evaluating the second) and Program State (the big picture: how already evaluated statements affected the analysis).

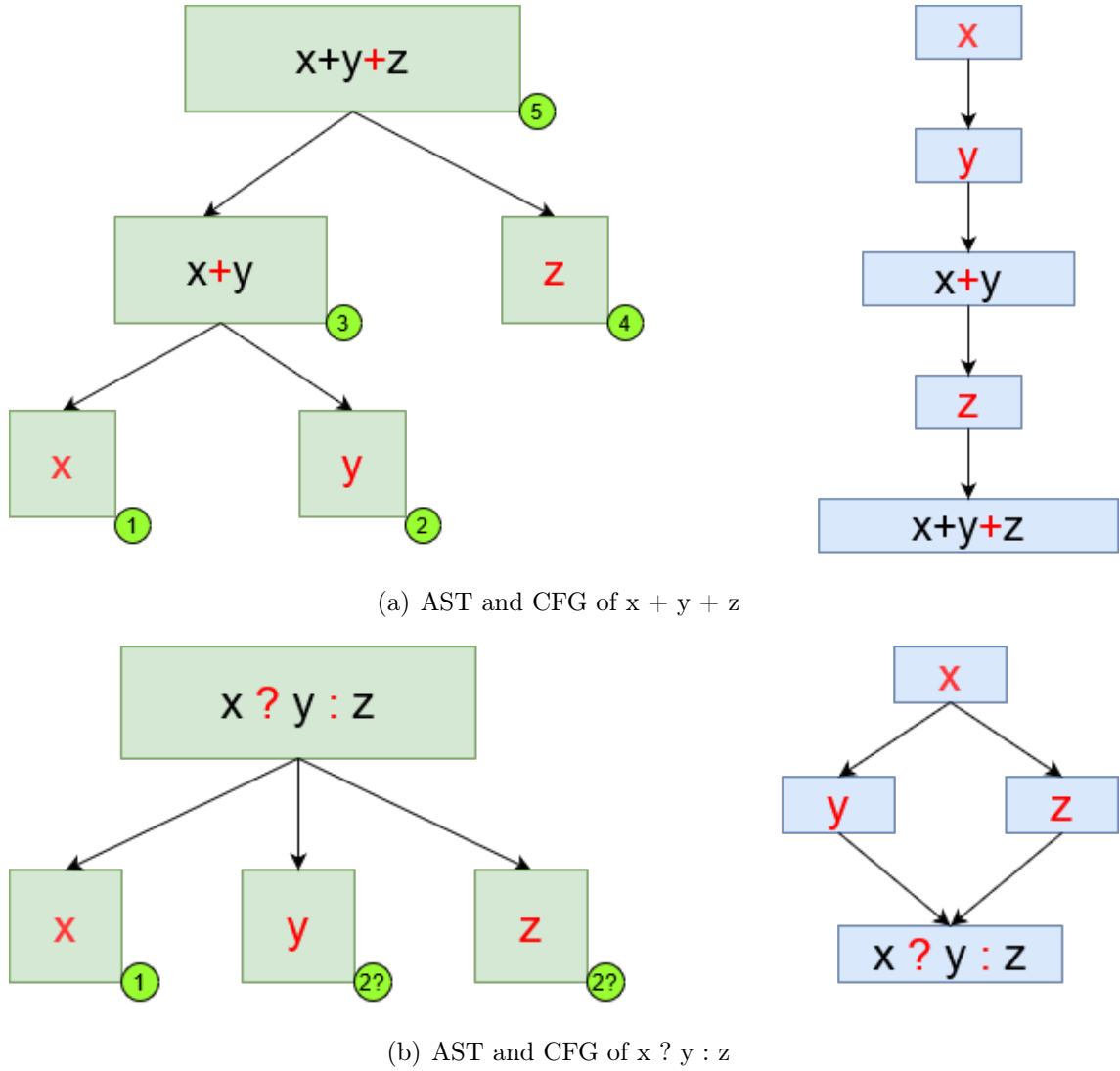


Figure 3.1: Comparison of AST and CFG

### 3.3.2 Checkers

Checkers are independent analyzer modules. They inherit from a template class `Checker`; the template parameters are types of events to which checker is subscribed. For example, if a checker is interested in an event that occurs after a function call is processed, it can inherit from `Checker<check::PostCall>` and implement `checkPostCall` to inspect the event, issue a warning if necessary, or provide additional information to the analysis (checkers are active participants in the graph building process). The Checker Developer's Guide [25] contains a list of all such callbacks as well as examples of their use.

### 3.3.3 Custom Program States

During the analysis there is no guarantee about the order in which the program will be explored, or even that all possible paths will be explored; As a result, when checkers need to keep track of information specific to their work, it can not be stored within a single checker class. Instead, they add the data to `ProgramState`. There are few macros designed for this purpose:

- `REGISTER_TRAIT_WITH_PROGRAMSTATE` – Used when the state information is a single value.
- `REGISTER_LIST_WITH_PROGRAMSTATE` – Used when the state information is a list of values.
- `REGISTER_SET_WITH_PROGRAMSTATE` – Used when the state information is a set of values.
- `REGISTER_MAP_WITH_PROGRAMSTATE` – Used when the state information is a map from a key to a value.

All the above data structures come with convenient getter and setter functions. However, since `ProgramState` is immutable, whenever we modify the data inside it, a copy of the state with the change applied is created. By calling the `CheckerContext::addTransition` function, the updated state must be returned to the analyzer core.

## 3.4 Implementation

This section describes the implementation of `alpha.security.cert.pos.34c` and `alpha.security.cert.env.InvalidPtr` checkers.

### 3.4.1 `alpha.security.cert.pos.34c`

The idea behind this checker is straightforward, and as one of the LLVM reviewers put it, "the source code reads so easily, we might as well put it as the official CERT rule description". We subscribe for function call events, determine if it is `putenv`, and inspect its argument, see figure 3.2.

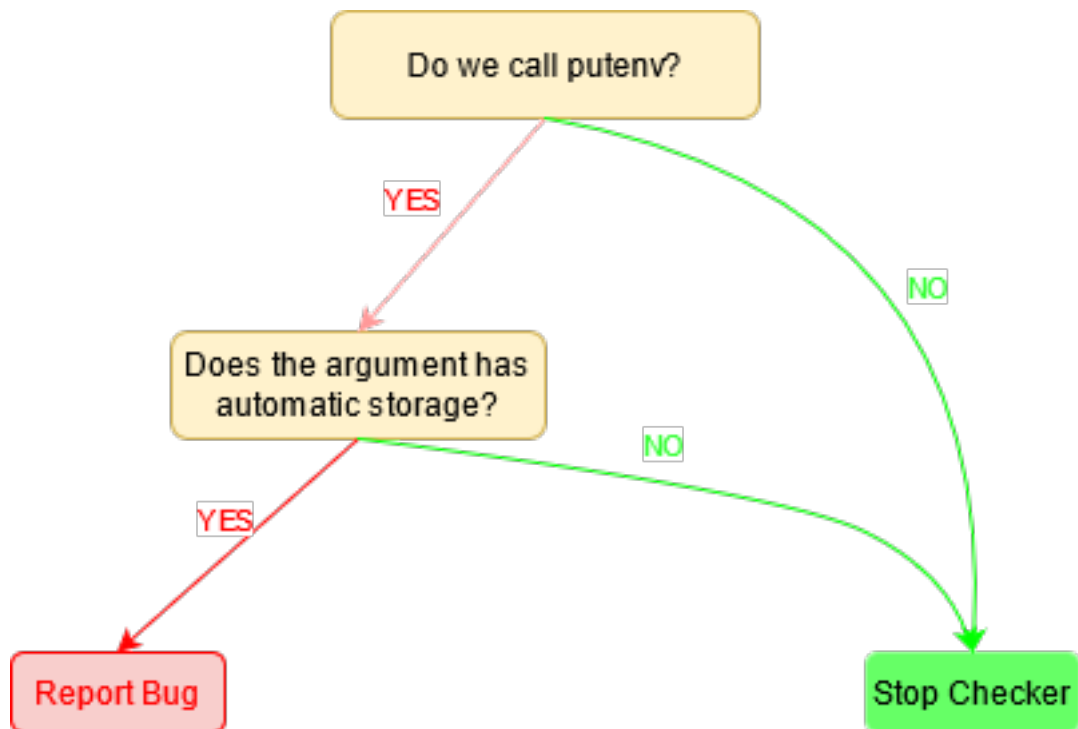


Figure 3.2: PutenvWithAuto checker logic

We use `void checkPostCall(const CallEvent &Call, CheckerContext &C)` callback, which passes control to the checker after any function call.

`CallEvent::isCalled()` function takes `CallDescription` type as an argument and determines whether the called function matches one in the argument. To accomplish this, we first represent `putenv` as `CallDescription`.

```

1 class PutenvWithAutoChecker : public Checker<check::PostCall> {
2 private:
3     // ...
4     const CallDescription Putenv{"putenv", 1};
5 public:
6     void checkPostCall(const CallEvent &Call, CheckerContext &C)
7         const;
8 };
9
10 void PutenvWithAutoChecker::checkPostCall(const CallEvent &Call,
11                                             CheckerContext &C) const
12 {
13     if (!Call.isCalled(Putenv))
14         return;
15     // ...
16 }

```

If the called function matches `putenv`, we proceed to the argument, which is represented as `SVal`. We find the *Memory Space Region* of this `SVal` and compare it to the `StackSpaceRegion`, the base class of everything stored on a stack.

```

1 void PutenvWithAutoChecker::checkPostCall(const CallEvent &Call,
2                                           CheckerContext &C) const
3 {
4     // ...
5     SVal ArgV = Call.getArgSVal(0);
6     const MemSpaceRegion *MSR =
7         ArgV.getAsRegion()->getMemorySpace();
8     if (!isa<StackSpaceRegion>(MSR))
9         return;
10    // ...
11 }

```

If we passed the both return statements in the preceding code snippets, we have violated POS34-C, as described in section 1.2, and we must report the bug.

This is accomplished by creating `PathSensitiveBugReport`, which takes three parameters:

1. The type of a bug, instance of `BugReport` class.
2. Short descriptive error message.
3. `ExplodedNode`, the context in which the problem happened. This includes the location of the bug and the current state.

After `BugReport` is created, it must be passed to the analyzer core by `CheckerContext::emitReport` function.

```

1 class PutenvWithAutoChecker : public Checker<check::PostCall> {
2 private:
3     BugType BT{this, "'putenv' function should not be called with
4         auto variables", categories::SecurityError};
5     // ...
6 };
7 void PutenvWithAutoChecker::checkPostCall(const CallEvent &Call,
8                                           CheckerContext &C) const
9 {

```

```
9      // ...
10    StringRef ErrorMsg = "The 'putenv' function should not be
        called with arguments that have automatic storage";
11    ExplodedNode *N = C.generateErrorNode();
12     auto Report = std::make_unique<PathSensitiveBugReport>(BT,
        ErrorMsg, N);
13     C.emitReport(std::move(Report));
14 }
```

As a next step, we register the new checker by including two boilerplate functions to the source code:

```
1 void ento::registerPutenvWithAuto(CheckerManager &Mgr) {
2     Mgr.registerChecker<PutenvWithAutoChecker>();
3 }
4
5 bool ento::shouldRegisterPutenvWithAuto(const LangOptions &) {
6     return true;
7 }
```

By adding it to `lib/StaticAnalyzer/Checkers/CMakeLists.txt`, the source code file is made visible to CMake.

Finally, inside `include/clang/StaticAnalyzer/Checkers/Checkers.td` we select a package for the checker. The "CERT" package was created as a child of "SecurityAlpha," and it has two children, "POS" and "ENV" for our two checkers, respectively, as shown in the figure 3.3. One can verify that new checker was added by checking available checkers: `clang -cc1 -analyzer-checker-help`.

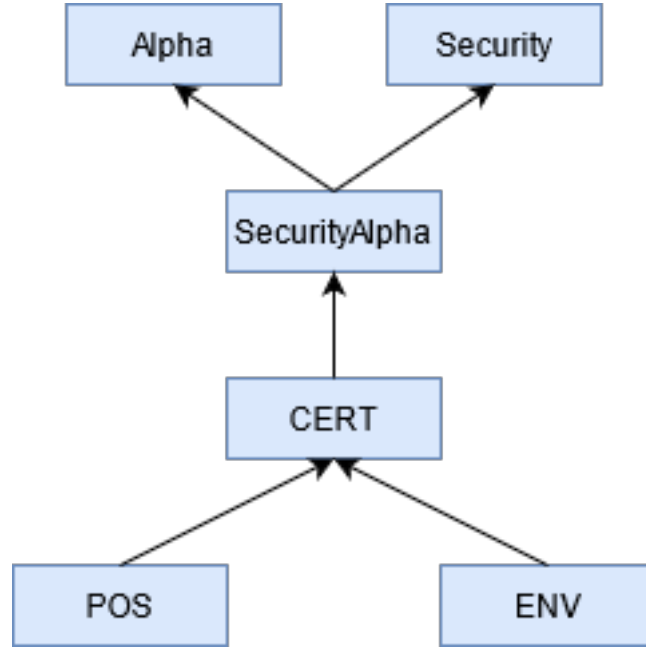


Figure 3.3: Checker packages tree

### 3.4.2 `alpha.security.cert.env.InvalidPtr`

A close examination of rules ENV31-C in section 1.2.2 and ENV34-C in section 1.2.3 reveals that they are very similar. In both cases, some sequence of events invalidates pointers, and subsequent dereference of these invalid pointers results in unintended behavior. Furthermore, there are several scenarios in which a pointer escapes analysis, such as when it is passed to a function that cannot be analyzed. To further reduce false negatives we will add one more heuristic and issue a warning when an invalidated pointer is used as an argument to a function that is not conservatively evaluated.

Assuming we have a collection of invalidated pointers, we can use the same logic to check for usage and issue a warning in both cases, as shown in figure 3.4. This inspires us to create a single checker that handles both rules.

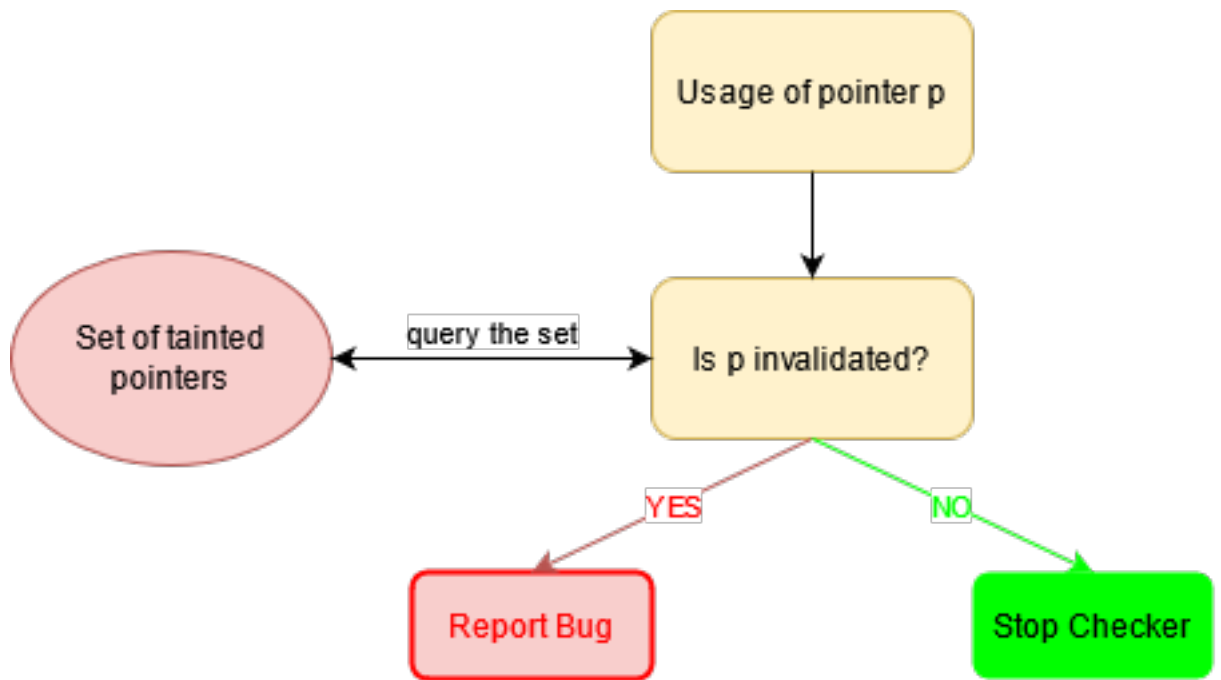


Figure 3.4: High level logic of InvalidPtr checker

### Invalidating pointer – ENV31-C

The first step is to get the `envp` argument of `main`, if it exists. For this we use `checkBeginFunction`, which is called when the core begins to analyze a function. We check if function is `main` and store third argument inside the state.

```

1 // Stores the region of the environment pointer of 'main' (if
  present).
2 // Note: This pointer has type 'const MemRegion *'
3 REGISTER_TRAIT_WITH_PROGRAMSTATE(EnvPtrRegion, const void *)
4
5 void InvalidPtrChecker::checkBeginFunction(CheckerContext &C) const
6 {
7     if (!C.inTopFrame())
8         return;
9
10    const auto *FD = dyn_cast<FunctionDecl>(C.getLocationContext()->
      getDecl());
11    if (!FD || FD->param_size() != 3 || !FD->isMain())
12        return;
13
14    ProgramStateRef State = C.getState();

```



```

14  const MemRegion *EnvpReg =
15      State->getRegion(FD->parameters()[2], C.getLocationContext())
16      ;
17  // Save the memory region pointed by the environment pointer
18  State = State->set<EnvPtrRegion>(
19      reinterpret_cast<void *>(const_cast<MemRegion *>(EnvpReg)));
20  C.addTransition(State);
21  }

```

The next step is to move the envp pointer to an invalidated set, if an environment modifying function is called, Such functions are stored in `CallDescriptionMap`, which is a map from `CallDescription` to their handler functions. Similarly to POS34-C, we use `checkPostCall` to listen for function calls and, if it is found in our map, we call the handler.

```

1  void EnvpInvalidatingCall(const CallEvent &Call, CheckerContext &
2      C) const;
3  using HandlerFn = void (InvalidPtrChecker::*)(const CallEvent &
4      Call, CheckerContext &C) const;
5  const CallDescriptionMap<HandlerFn> EnvpInvalidatingFunctions = {
6      {"setenv", 3}, &InvalidPtrChecker::EnvpInvalidatingCall},
7      {"unsetenv", 1}, &InvalidPtrChecker::EnvpInvalidatingCall},
8      {"putenv", 1}, &InvalidPtrChecker::EnvpInvalidatingCall},
9      {"_putenv_s", 2}, &InvalidPtrChecker::EnvpInvalidatingCall},
10     {"_wputenv_s", 2}, &InvalidPtrChecker::EnvpInvalidatingCall
11     },
12 };
13
14 void InvalidPtrChecker::checkPostCall(const CallEvent &Call,
15     CheckerContext &C) const {
16     // Check if function invalidates 'envp' argument of 'main'
17     if (const auto *Handler = EnvpInvalidatingFunctions.lookup(Call
18         ))
19         (this->**Handler)(Call, C);
20     // ...
21 }
22
23 void InvalidPtrChecker::EnvpInvalidatingCall(const CallEvent &

```

```
    Call, CheckerContext &C) const {  
19    // check if EnvPtrRegion exists  
20    // add it to the set of invalidated pointers  
21 }
```

### Invalidating pointer – ENV34-C

Aside from the five functions mentioned in the rule, many non-reentrant C library functions suffer from a similar problem. This part of the checker was designed with genericness in mind, so that simply adding a new entry to the `CallDescriptionMap` is sufficient to extend the checker.

```
1  const CallDescriptionMap<EvalFn>  
    PreviousCallInvalidatingFunctions = {  
2      {"getenv", 1}, &InvalidPtrChecker::evalGetenv},  
3      {"setlocale", 2}, &InvalidPtrChecker::evalSetlocale},  
4      {"strerror", 1}, &InvalidPtrChecker::evalStrerror},  
5      {"localeconv", 0}, &InvalidPtrChecker::evalLocaleconv},  
6      {"asctime", 1}, &InvalidPtrChecker::evalAsctime},  
7  };
```

We create a custom map in which the keys are function names and the values are the results of the most recent call to this function:

```
REGISTER_MAP_WITH_PROGRAMSTATE(PreviousCallResultMap, const char *, const  
MemRegion *)
```

On each call to the problematic function, we move the previous value (if it exists) from `PreviousCallResultMap` to the invalidated set and update the map with the new memory region.

### Reporting a bug

Every time a specific memory location is accessed, the `checkLocation` callback is triggered. Inside it we check if accessed memory is invalidated and report dereference of an invalidated pointer if it is.

```
1 void InvalidPtrChecker::checkLocation(SVal Loc, bool isLoad,
2     const Stmt *S,
3     CheckerContext &C) const {
4     ProgramStateRef State = C.getState();
5     // Ignore memory operations involving 'non-invalidated' locations
6     .
7     const MemRegion *InvalidatedSymbolicBase =
8         findInvalidatedSymbolicBase(State, Loc.getAsRegion());
9     if (!InvalidatedSymbolicBase)
10         return;
11     ExplodedNode *ErrorNode = C.generateNonFatalErrorNode();
12     if (!ErrorNode)
13         return;
14
15     auto Report = std::make_unique<PathSensitiveBugReport>(
16         BT, "dereferencing an invalid pointer", ErrorNode);
17     C.emitReport(std::move(Report));
18 }
```

We also issue a warning whenever an invalidated pointer is passed as an argument to a non-conservatively analyzed function, as mentioned in 3.4.2. We do this within `checkPostCall` by traversing the array of arguments of a function and querying `InvalidatedPointerSet` for each of them.

## 3.5 Testing

Clang Static Analyzer checker testing consists of two parts: lit testing (see 3.5.1) and running analysis on large code bases to find real world bug examples and, more importantly, to see if the analyzer breaks (see 3.5.2).

### 3.5.1 Lit

The idea behind lit testing is simple: we create test files with faulty source code and specify which lines should generate a warning or a note. Clang's `-verify` flag performs this type of verification. Consider the following example:

```
1 void getenv_test() {
2     char *p1, *p2, *p3;
3
4     p1 = getenv("VAR1");
5     *p1; // no-warning
6
7     p1 = getenv("VAR2"); // expected-note{{previous function call was
8         here}}
9
10    p3 = getenv("VAR3"); // expected-note{{'getenv' call may
11        invalidate the the result of the previous 'getenv'}}
12
13    p2 = getenv("VAR4");
14
15    *p1; // expected-warning{{dereferencing an invalid pointer}}
16 }
```

Compliant code must also be tested; typically, a "no-warning" comment is written on lines that may be problematic; however, this is just a convention among analyzer developers and has no actual meaning behind it.

`expected-note` and `expected-warning` are sometimes referred as verify comments. It is also possible to pass them some optional arguments, such as the distance between the comment and the actual warning, or the number of times the analyzer should throw an error. For a better illustration, consider the following slightly modified version of the preceding example:

```
1 void getenv_test1() {
2     char *p1, *p2, *p3;
3
4     p1 = getenv("VAR1");
5     *p1; // no-warning
6
7     p1 = getenv("VAR2");
8     // expected-note@-1{{previous function call was here}}
9
10    p3 = getenv("VAR3");
11    // expected-note@-1{{'getenv' call may invalidate the the result
12        of the previous 'getenv'}}
13
14    p2 = getenv("VAR4");
15 }
```

```
14 // expected-note@+1{{dereferencing an invalid pointer}}
15 *p;
16 // expected-warning@-1{{dereferencing an invalid pointer}}
17 }
```

## LLVM Integrated Tester

llvm-lit is a lightweight tool for LLVM black box testing.

Each lit test file should begin with the following comment: `// RUN: <command>`, where `command` is argument for `llvm-lit` to run in terminal.

```
1 // RUN: %clang_analyze_cc1 \
2 // RUN: -analyzer-checker=alpha.security.cert.env.InvalidPtr\
3 // RUN: -analyzer-output=text -verify %s
```

`clang_analyze_cc1` is a macro that replaces analyzer invocation. On the third line, the output type `text` is specified to check for issued notes as well as warnings (see 2.3.1), and the `-verify` flag is enabled.

ENV31-C (section 1.2.2) states that pointers can be invalidated by operations that modify the environment, but there are a few ways to do so. To avoid code duplication and having a similar test for each env modifying function, we use the macro `ENV_INVALIDATING_CALL`, which takes different values on different lit invocations. This is the test file for this rule:

```
1 // RUN: %clang_analyze_cc1 -analyzer-output=text %s\
2 // RUN: -analyzer-checker=core,alpha.security.cert.env.InvalidPtr\
3 // RUN: -verify=putenv,common\
4 // RUN: -DENV_INVALIDATING_CALL="putenv(\"X=Y\")"
5 //
6 // RUN: %clang_analyze_cc1 -analyzer-output=text %s \
7 // RUN: -analyzer-checker=core,alpha.security.cert.env.InvalidPtr\
8 // RUN: -verify=putenvs,common\
9 // RUN: -DENV_INVALIDATING_CALL="_putenv_s(\"X\", \"Y\")"
10 //
11 // RUN: %clang_analyze_cc1 -analyzer-output=text %s\
12 // RUN: -analyzer-checker=core,alpha.security.cert.env.InvalidPtr\
13 // RUN: -verify=wputenvs,common\
14 // RUN: -DENV_INVALIDATING_CALL="_wputenv_s(\"X\", \"Y\")"
15 //
```

```
16 // RUN: %clang_analyze_cc1 -analyzer-output=text %s\  
17 // RUN: -analyzer-checker=core,alpha.security.cert.env.InvalidPtr\  
18 // RUN: -verify=setenv,common\  
19 // RUN: -DENV_INVALIDATING_CALL="setenv(\"X\", \"Y\", 0)"  
20 //  
21 // RUN: %clang_analyze_cc1 -analyzer-output=text %s\  
22 // RUN: -analyzer-checker=core,alpha.security.cert.env.InvalidPtr\  
23 // RUN: -verify=unsetenv,common\  
24 // RUN: -DENV_INVALIDATING_CALL="unsetenv(\"X\")"  
25  
26 // ...  
27 // Function and type definitions  
28 // ...  
29  
30 void call_env_invalidating_fn(char **e) {  
31     ENV_INVALIDATING_CALL;  
32     // putenv-note@-1 5 {'putenv' call may invalidate the  
33         environment parameter of 'main'}}  
34     // putenvs-note@-2 5 {'_putenv_s' call may invalidate the  
35         environment parameter of 'main'}}  
36     // wputenvs-note@-3 5 {'_wputenv_s' call may invalidate the  
37         environment parameter of 'main'}}  
38     // setenv-note@-4 5 {'setenv' call may invalidate the  
39         environment parameter of 'main'}}  
40     // unsetenv-note@-5 5 {'unsetenv' call may invalidate the  
41         environment parameter of 'main'}}  
42  
43     *e;  
44     // common-warning@-1 {{dereferencing an invalid pointer}}  
45     // common-note@-2 {{dereferencing an invalid pointer}}  
46 }  
47  
48 int main(int argc, char *argv[], char *envp[]) {  
49     char **e = envp;  
50     *e; // no-warning  
51     e[0]; // no-warning  
52     *envp; // no-warning  
53     call_env_invalidating_fn(e);  
54     // common-note@-1 5 {{Calling 'call_env_invalidating_fn'}}
```

```

50 // common-note@-2 4 {{Returning from 'call_env_invalidating_fn'}}
51
52 *e;
53 // common-warning@-1 {{dereferencing an invalid pointer}}
54 // common-note@-2 {{dereferencing an invalid pointer}}
55
56 *envp;
57 // common-warning@-1 2 {{dereferencing an invalid pointer}}
58 // common-note@-2 2 {{dereferencing an invalid pointer}}
59
60 fn_without_body(e);
61 // common-warning@-1 {{use of invalidated pointer 'e' in a
    function call}}
62 // common-note@-2 {{use of invalidated pointer 'e' in a function
    call}}
63
64 fn_with_body(e); // no-warning
65 }

```

It is worth noting that the `verify` flag accepts optional arguments and asserts only those comments on each run.

Following command can be used to run all POS34 and InvalidPtr tests:

```
llvm-lit path/to/llvm-project/clang/test/Analysis/cert -a
```

Test case	Source file, line	Description
putenv	pos34-c-fp-suppression.cpp, 13	no warning on extern variable
putenv	pos34-c-fp-suppression.cpp, 18	no warning on non-auto variable
putenv	pos34-c-fp-suppression.cpp, 27	no warning on non-auto variable
putenv	pos34-c-fp-suppression.cpp, 38	false positive, marked as todo
putenv	pos34-c.cpp, 16	warn on volatile variable, example from CERT
putenv	pos34-c.cpp, 31	no warning on static variable, example from CERT
putenv	pos34-c.cpp, 42	no warning on heap variable, example from CERT

Table 3.1: Summary of automated tests for POS34-C

Test case	Source file, line	Description
putenv, putenv_s, wputenv_s, setenv, unsetenv	env31-c.c, 46	warn dereference of invalidated pointer
putenv, putenv_s, wputenv_s, setenv, unsetenv	env31-c.c, 60	warn dereference of invalidated pointer after returning from function call
putenv, putenv_s, wputenv_s, setenv, unsetenv	env31-c.c, 64	warn dereference of alias of invalidated pointer
putenv, putenv_s, wputenv_s, setenv, unsetenv	env31-c.c, 68	warn non-inlined function call with invalidated pointer
putenv, putenv_s, wputenv_s, setenv, unsetenv	env31-c.c, 72	no warning on inlined function call with invalidated pointer

Table 3.2: Summary of automated tests for ENV31-C



Test case	Source file, line	Description
getenv	env34-c.c, 31	no warning if second getenv result is assigned to same pointer
getenv	env34-c.c, 39	no warning dereference of getenv return pointer
getenv	env34-c.c, 44	warn dereference of invalidated pointer
getenv	env34-c.c, 62	warn dereference of invalidated pointer
getenv	env34-c.c, 76	warn dereference of the first invalidated pointer after third getenv call
getenv	env34-c.c, 90	warn dereference of the second invalidated pointer after third getenv call
getenv	env34-c.c, 108	warn dereference of invalidated pointer
getenv	env34-c.c, 118	warn dereference of invalidated pointer
getenv	env34-c.c, 118	warn function call with invalidated pointer
getenv	env34-c.c, 157	warn dereference of invalidated array member pointer
getenv	env34-c.c, 167	no warning if pointer escapes analysis
getenv	env34-c.c, 171	warn function call with getenv calls as arguments
getenv	env34-c.c, 179	warn dereference of pointer that was invalidated outside function
getenv	env34-c.c, 210	warn dereference of pointer with conditional flags

Table 3.3: Summary of automated tests for ENV34-C

Test case	Source file, line	Description
setlocale	env34-c.c, 222	no warning if second setlocale result is assigned to same pointer
setlocale	env34-c.c, 227	warn dereference of invalidated pointer
setlocale	env34-c.c, 248	warn dereference of invalidated pointer, flow sensitive
strerror	env34-c.c, 261	no warning if second strerror result is assigned to same pointer
strerror	env34-c.c, 266	warn dereference of invalidated pointer
strerror	env34-c.c, 296	warn dereference of invalidated pointer, flow sensitive
asctime	env34-c.c, 310	warn dereference of invalidated pointer
localeconv	env34-c.c, 321	warn dereference of invalidated pointer
localeconv	env34-c.c, 330	false negative, marked as todo

Table 3.4: Summary of automated tests for ENV34-C

### 3.5.2 Running analysis on large code bases

The checker will then be run on several large open source projects to assess its usefulness and ensure that new changes do not cause the analyzer to crash. We use CodeChecker (section 2.3.4) to run the analysis and examine the reports.

#### Projects used for testing

These open source code bases were analyzed:

- **LLVM** – LLVM [26] is one of the largest open source projects written in C++, with millions of lines of code. It is common practice to run new checkers on LLVM itself as a form of dogfooding. Analysis returned no results for POS34-C, ENV31-C, or ENV34-C and completed successfully without any crashes.
- **SQLite** – SQLite [27] is world’s most used SQL database engine written in C. Running our checkers did not break analysis and resulted in three true positive reports.

- **FFmpeg** – FFmpeg [28] is a popular tool for recording, converting, and streaming audio and video. Checkers discovered seven violations of ENV34-C, all of which were true positives. One of them is depicted in figure 3.5.
- **Git** – Git [24] is the world’s most popular version control system. It is written in C and heavily relies on environment variables, making it an excellent candidate for testing our checkers. Analyzer completed successfully and generated 54 reports, the majority of which were true positives. Figure 3.6 is an example.

All of the above-mentioned findings will be reported to the appropriate project maintainers.

Curl, memcached, nginx, PostgreSQL, Redis, tmux, BitCoin, OpenSSL, Xerces, and VIM were also examined. The analysis finished without any crashes, but no reports were generated, which was confirmed by inspecting the source code.

Project	Lines of code	#Findings	#TruePos	#FalsePos
SQLite	1.0 million	3	3	0
FFmpeg	1.9 million	7	7	0
Git	1.4 million	54	51	3
LLVM	21.2 million	0	0	0
Curl	512.8k	0	0	0
memcached	50.2k	0	0	0
PostgreSQL	3.5 million	0	0	0
Redis	306.1k	0	0	0
tmux	146.4k	0	0	0
Bitcoin	667.7k	0	0	0
OpenSSL	1.3 million	0	0	0
Xerces	361.8k	0	0	0
vim	1.6 million	0	0	0

Table 3.5: Summary of testing checkers on open source projects

```

File: /home/zuka/testDir/ffmpeg/fftools/cmdutils.c
Checker name: alpha.security.cert.env.InvalidPtr
2048                                     const char *codec_name);
2049 {
2050     FILE *f = NULL;
2051     int i;
2052     const char *base[3] = { getenv("FFMPEG_DATADIR"),
                             1 previous function call result assigned here >
                             3 < Initialized here >
                             getenv("HOME"),
                             2 < 'getenv' call may invalidate the the result of the previous 'getenv' >
                             FFMPEG_DATADIR, };
2053
2054     if (is_path) {
2055         4 < Assuming 'is_path' is 0 >
2056         av_strncpy(filename, preset_name, filename_size);
2057         f = fopen(filename, "r");
2058     } else {
2059         #if HAVE_GETMODULEHANDLE && defined(_WIN32)
2060             char datadir[MAX_PATH], *ls;
2061             base[2] = NULL;
2062
2063             if (GetModuleFileNameA(GetModuleHandleA(NULL), datadir, sizeof(datadir) - 1))
2064             {
2065                 for (ls = datadir; ls < datadir + strlen(datadir); ls++)
2066                     if (*ls == '\\') *ls = '/';
2067
2068                 if (ls = strrchr(datadir, '/'))
2069                 {
2070                     *ls = 0;
2071                     strcat(datadir, "/ffpresets", sizeof(datadir) - 1 - strlen(datadir));
2072                     base[2] = datadir;
2073                 }
2074             }
2075         #endif
2076         for (i = 0; i < 3 && !f; i++) {
2077             5 < The value 0 is assigned to 'i' >
2078             6 < Entering loop body >
2079             if (!base[i])
2080                 7 < Assuming the condition is false >
2081                 continue;
2082             snprintf(filename, filename_size, "%s%s/%s.ffpreset", base[i],
2083                     i != 1 ? "" : "/.ffmpeg", preset_name);
2084             f = fopen(filename, "r");
2085             if (!f && codec_name) {

```

Figure 3.5: Invalidated pointer usage in FFmpeg

```

679 static char* getdir(void)
680 {
681     struct strbuf buf = STRBUF_INIT;
682     char *pathinfo = getenv("PATH_INFO");
683     char *root = getenv("GIT_PROJECT_ROOT");
684     char *path = getenv("PATH_TRANSLATED");
685
686     if (root && *root) {
687         4 < Entered call from 'cmd_main' >
688         5 < previous function call result assigned here >
689         6 < 'getenv' call may invalidate the the result of the previous 'getenv' >
690         7 < Assuming 'root' is non-null >
691         8 < dereferencing an invalid pointer >

```

Figure 3.6: Invalidated pointer dereference in Git

# Chapter 4

## Conclusion and future work

This thesis presented a solution to automate the detection of three different types of bugs in C and C++ codebases. We developed two checkers for Clang Static Analyzer, an open-source static analysis tool. The project was a success, and it met all of the original specifications. One of the checkers has already passed the LLVM code review, demonstrating its high quality, and runs on millions of devices across the world as part of the Clang. Thesis documentation can be used as a guide by anyone who wishes to develop Clang Static Analyzer checkers.

Both checkers can be improved. The InvalidPtr checker's short-term goal would be to successfully complete the LLVM review, later investigation should follow to extend it with more non-reentrant functions that suffer from similar problem, whereas the POS34 checker's goal would be to remove it from the so-called "alpha" state, i.e. enable it by default for all Clang Static Analyzer runs. This could be accomplished by reducing the number of false positives, for example, by warning on the last `putenv()` call on the execution path through the current stack frame.

Overall, this project is not only my favorite topic in program analysis, but it is also an excellent choice for demonstrating knowledge gained throughout my bachelor's degree studies.

## Acknowledgements

I would like to thank the following people, without whom I would not have been able to complete this thesis, and without whom I would not have made it through

my bachelor degree!

First and foremost, I'd like to thank my supervisor, Prof. Zoltán Porkoláb, for assisting me in selecting a thesis topic and guiding me through the process of writing thesis work.

I would also like to extend my deepest gratitude to Balázs Benics (ELTE, Ericsson) and Csaba Dabis (ELTE) for assisting with the design of the checkers, as well as answering all of my questions during the development phase, dedicating time for one-on-one meetings even on weekends, debugging the code with me, and suggesting how to further polish the work for the LLVM submission.

Special thanks to Ericsson's CodeChecker team for their valuable suggestions during the early phase of the checkers. And to my team at Ericsson – CI Joe, for giving me the flexibility and time needed to work on this thesis.

Many thanks to Prof. Zoltán Gera for mentoring and introducing me to the Clang in Security Checker Development lab, as well as for teaching Imperative Programming during the first semester of my bachelor studies, the subject that served as the foundation for my C++ and C knowledge.

I would also like to acknowledge Prof. Viktoria Zsók, who first introduced me to doing research early at my bachelor studies. Writing a thesis was made much easier because of previous experience.

I am also grateful to LLVM reviewers: Artem Dergachev (Apple Inc.), Kristóf Umann (ELTE, Ericsson), and Aaron Ballman (Intel Corp.), for their insightful comments on the code and for making it possible that part of my thesis runs on millions of devices worldwide as part of Clang.

# Appendix A

## List of created and modified files

Changes were made only in `llvm-project/clang` directory.

### A.1 List of modified files for POS34

- `clang/docs/analyzer/checkers.rst`
- `clang/include/clang/StaticAnalyzer/Checkers/Checkers.td`
- `clang/include/clang/StaticAnalyzer/Core/BugReporter/CommonBugCategories.h`
- `clang/lib/StaticAnalyzer/Checkers/CMakeLists.txt`
- `clang/lib/StaticAnalyzer/Core/CommonBugCategories.cpp`

### A.2 List of added files for POS34

- `clang/lib/StaticAnalyzer/Checkers/cert/PutenvWithAutoChecker.cpp`
- `clang/test/Analysis/cert/pos34-c-fp-suppression.cpp`
- `clang/test/Analysis/cert/pos34-c.cpp`

### A.3 List of modified files for InvalidPtr

- `clang/docs/analyzer/checkers.rst`

- clang/include/clang/StaticAnalyzer/Checkers/Checkers.td
- clang/lib/StaticAnalyzer/Checkers/CMakeLists.txt

## **A.4 List of added files for InvalidPtr**

- clang/lib/StaticAnalyzer/Checkers/cert/InvalidPtrChecker.cpp
- clang/test/Analysis/cert/env31-c.cpp
- clang/test/Analysis/cert/env34-c.cpp
- clang/test/Analysis/cert/env34-c-cert-examples.c



# Bibliography

- [1] *Software problems led to system failure at Dhahran, Saudi Arabia*. Washington DC: Information Management and Technology Division, US General Accounting Office, 1992, p. 16.
- [2] Matthew Philips. *Knight Shows How to Lose \$440 Million in 30 Minutes*. URL: <https://www.bloomberg.com/news/articles/2012-08-02/knight-shows-how-to-lose-440-million-in-30-minute> (visited on 05/13/2020).
- [3] Michael Krigsman. *IT failure at Heathrow T5: What really happened*. URL: <https://www.zdnet.com/article/it-failure-at-heathrow-t5-what-really-happened/> (visited on 05/13/2020).
- [4] *Cluster (spacecraft) Launch Failure*. URL: [https://en.wikipedia.org/wiki/Cluster\\_\(spacecraft\)#Launch\\_failure](https://en.wikipedia.org/wiki/Cluster_(spacecraft)#Launch_failure) (visited on 05/13/2020).
- [5] *Hearthbleed Bug*. URL: <https://heartbleed.com/> (visited on 05/13/2020).
- [6] *SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems*. 2016th ed. CERT, 2016.
- [7] *SEI CERT C++ Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems*. 2016th ed. CERT, 2016.
- [8] *POS34-C. Do not call putenv() with a pointer to an automatic variable as the argument*. URL: <https://wiki.sei.cmu.edu/confluence/x/6NYxBQ> (visited on 05/13/2020).
- [9] *ENV31-C. Do not rely on an environment pointer following an operation that may invalidate it*. URL: <https://wiki.sei.cmu.edu/confluence/x/5NUxBQ> (visited on 05/13/2020).

- [10] *ENV34-C. Do not store pointers returned by certain functions.* URL: <https://wiki.sei.cmu.edu/confluence/x/8tYxBQ> (visited on 05/13/2020).
- [11] Seth Kenlon. *What is POSIX?* 2019. URL: <https://opensource.com/article/19/7/what-posix-richard-stallman-explains> (visited on 05/13/2020).
- [12] *putenv function documentation IEEE Std 1003.1.* URL: <https://pubs.opengroup.org/onlinepubs/009696699/functions/putenv.html> (visited on 05/13/2020).
- [13] *environ - Global array to hold environment variables.* URL: <https://www.mksssoftware.com/docs/man5/environ.5.asp> (visited on 05/13/2020).
- [14] *The C Standard, J.5.1 [ISO/IEC 9899:2011] ISO/IEC. Programming Languages—C, 3rd ed (ISO/IEC 9899:2011).* Geneva, Switzerland: ISO, 2011..
- [15] *\_environ, \_wenviron.* URL: <https://docs.microsoft.com/en-us/cpp/c-runtime-library/environ-wenviron?view=msvc-160> (visited on 05/13/2020).
- [16] *Updated Field Experience With Annex K — Bounds Checking Interfaces.* URL: <http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1969.htm> (visited on 05/13/2020).
- [17] Ted Kremenek, Apple Inc. *Finding software bugs with the clang static analyzer.* 2008.
- [18] Zhongxing Xu, Ted Kremenek, Jian Zhang. “A memory model for static analysis of C programs”. In: *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation* (), pp. 535–548.
- [19] Anna Zaks, Jordan Rose. *LLVM Developers’ Meeting – Building a Checker in 24 hours.* 2012. URL: <https://llvm.org/devmtg/2012-11/Zaks-Rose-Checker24Hours.pdf> (visited on 05/13/2020).
- [20] *LLVM Project.* URL: <https://github.com/llvm/llvm-project> (visited on 05/13/2020).
- [21] *Ninja.* URL: <https://ninja-build.org/> (visited on 05/13/2020).

- [22] *scan-build*. URL: <https://clang-analyzer.llvm.org/scan-build.html> (visited on 05/13/2020).
- [23] *CodeChecker*. URL: <https://github.com/Ericsson/codechecker> (visited on 05/13/2020).
- [24] *Git*. URL: <https://git-scm.com/> (visited on 05/13/2020).
- [25] Artem Dergachev. *The Checker Developer's Guide*. URL: <https://github.com/haoNoQ/clang-analyzer-guide> (visited on 05/13/2020).
- [26] *The LLVM Compiler Infrastructure*. URL: <https://llvm.org/> (visited on 05/13/2020).
- [27] *SQLite*. URL: <https://www.sqlite.org/> (visited on 05/13/2020).
- [28] *FFmpeg*. URL: <https://www.ffmpeg.org/> (visited on 05/13/2020).

# List of Figures

1.1	Clang AST example . . . . .	7
1.2	How statements affect the program state . . . . .	8
1.3	Static Analyzer warning example . . . . .	10
2.1	POS34-C checker output . . . . .	14
2.2	POS34-C with diagnostics . . . . .	15
2.3	HTML output example . . . . .	16
2.4	scan-build output . . . . .	17
2.5	CodeChecker html output . . . . .	19
2.6	CodeChecker graphical user interface, list of reports . . . . .	20
2.7	CodeChecker graphical user interface, path diagnostics . . . . .	20
3.1	Comparison of AST and CFG . . . . .	24
3.2	PutenvWithAuto checker logic . . . . .	26
3.3	Checker packages tree . . . . .	29
3.4	High level logic of InvalidPtr checker . . . . .	30
3.5	Invalidated pointer usage in FFmpeg . . . . .	42
3.6	Invalidated pointer dereference in Git . . . . .	42

# List of Tables

2.1	Clang Static Analyzer system requirements . . . . .	12
3.1	Summary of automated tests for POS34-C . . . . .	37
3.2	Summary of automated tests for ENV31-C . . . . .	38
3.3	Summary of automated tests for ENV34-C . . . . .	39
3.4	Summary of automated tests for ENV34-C . . . . .	40
3.5	Summary of testing checkers on open source projects . . . . .	41