# Verification of C Buffer Overflows in C Programs

Andreea Bican
Faculty of Automatic Control and Computers
University POLITEHNICA of Bucharest
Bucharest, Romania
andreea.bican@stud.acs.upb.ro

Răzvan Deaconescu
Faculty of Automatic Control and Computers
University POLITEHNICA of Bucharest
Bucharest, Romania
razvan.deaconescu@cs.pub.ro

Wei Ngan Chin
School of Computing
National University of Singapore
Singapore, Singapore
chinwn@comp.nus.edu.sg

Quang-Trung Ta
School of Computing
National University of Singapore
Singapore, Singapore
taqt@comp.nus.edu.sg

*Abstract*— **Buffer overflow attacks are a persisting security threat in C programs. The C Standard library provides functions for string handling that lack any bound checks. This paper presents a static approach for buffer overflow detection by identifying the likely vulnerabilities through an analysis of the source code. We defined a set of predicates, based on the function's specifications, that determine whether the operation is safe or not. This paper describes an implementation of the approach as an extension of HIP/SLEEK, an automated verification system based on the separation logic. The static buffer overflow detector proved to have good results even in tricky cases, such as pointer aliasing and overlapping memory.**

***Keywords—verification, buffer overflows, code analysis***

## I. Introduction

Buffer overflow attacks are an important and persistent security problem. Programs written in C are particularly vulnerable to those attacks. Space and performance were more important design considerations for C than safety. Hence, the language allows direct pointer manipulations without any bounds checking. The C Standard library includes many functions that are unsafe if they are not used carefully. The purpose of this project is to implement an automated formal detection of string buffer overflow vulnerabilities in C programs.

The most reliable way to avoid or prevent a vulnerability is to use automatic protection at the language level. This sort of protection, however, cannot be applied to legacy code, and often technical, business, or cultural constraints call for a vulnerable language.

In this paper we propose an approach for buffer overflow detection in source code, primarily aiming for providing a bounds checking mechanism for the string manipulation functions from C Standard library. It performs a static analysis on the source code that is able to detect the function calls that result in buffer overflows, but also the parts that may be vulnerable to an attack in case the size of the input is not checked.

The implementation is based on the separation logic for reasoning the program correctness. Each string is treated as a separate memory area via an explicit separation of structural properties. On the program side, we instrumented the program to be verified with explicit predefined operations over the allocated variables. Each verification is executed at function level and tries to prove that based on a set of preconditions, the postconditions hold. This way, the memory usage of the original program is mimicked. The instrumented programs are known as memory-aware programs.

In this we paper we use static analysis by employing HIP/SLEEK[6], a automated verification for the functional correctness of heap manipulating programs. HIP is a system based on the separation logic, able to verify the specifications (user defined predicates). It constructs a set of proof obligations in the form of formula implications which are sent to the SLEEK, the separation logic prover.

Our objectives are:

- We aim for a static verification that is able to detect buffer overflows on any string allocated. This implies that results will not depend on the memory where the data is stored, but on the properties of it. More specific, any detection mechanism should provide same results on any stack, heap or global allocated string.
- We aim for defining "contracts" that have to be validated for any unsafe string manipulation function defined in C Standard library. This includes the following: strcpy, strncpy, strcat, strncat, sprintf, snprintf, gets, fgets, scanf.

- We aim for a tool that is easy to use and does not require any modification to the source code from the user.
- We aim to use the separation logic for defining memory areas that can handle not only integer constraints, but also content checks. The idea is to be able to understand at a minimal level the content that can influence the behavior of a string manipulation (e.g. the existence/non-existence of a null terminating character).
- We aim to offer support for multidimensional arrays, with a focus on array of strings (char **). The latter has been exploited in the form of the command line arguments.

## II. Related Work

The formal approach of using annotated functions has been introduced and detailed in a few papers, but the problem of the static detection is undecidable. It works on heuristics and is, therefore, nor sound or complete. Several factors affect the consistency of the results: difficulty of bounds checking, pointers, aliasing, control flow and the unknown input before runtime.

Wagner et al. [5] formulated the buffer overflow detection problem as an integer constraint. It aims for scalability which comes at the cost of precision. It models each C string buffer as an abstract data structure represented by a set of two integers: number of bytes allocated and number of bytes used. The analysis is light and can be performed on big code, but it results in a high rate of false alarms and missed errors. It cannot handle pointers, aliasing, ignores the null terminator character and the whole control flow.

Larochelle and Evans [3] proposed an extension of the LCLint annotation-assisted static checking tool. The annotations, added on the libc string functions, specify the highest index that can be safely written to or read from. They claim their implementation to be "as fast as a compiler and nearly as easy to use". The LCLint tool, used for constraint specification, is both strict and flexible. The strictness comes from the imposed format of the annotations. The constraints can be conjoined, but there is no support for disjunction. As a result, not all the overflow scenarios can be reproduced and speculated in the annotations, thus many missed errors and false positives. This approach lacks the support for pointers, aliasing, ignores the null terminator character, but uses heuristics for the control flow.

Dor et. al[2] focused on string verification via static analysis though their work is limited to only several use cases.

### A. HIP/SLEEK

HIP/SLEEK is an automated verification system developed by Chin et al.[1][4] that supports finer levels of memory usage control through a set of specifications. This mechanism supports user-defined predicates that can be used to describe the properties of a wide range of data structures with different shapes (structural perspective) and sizes (numerical information).

The system consists of two separated entities.

HIP is one side of the proving framework. It is a predicate specification mechanism that can capture a wide range of data structures with different kinds of shapes, sizes and reachability properties. The verification system is based on separation logic, an extension of the Hoare logic, that supports reasoning about shared mutable data structures. HIP takes as an input a command and a set of pre and post conditions. It then derives the strongest postcondition upon termination of the command and checks if the strongest postcondition implies the declared postcondition.

SLEEK is an automated entailment prover which takes as an input a shape specification and a set of formulas. The shape specification is user-definable and provides flexibility in defining not only the shape of the data structure, but also the size and bag properties. The set of formulas is generated by the HIP predicate verification.

## III. Overview

The static analysis is implemented by combining traditional compiler data flow analysis with separation logic for reasoning the formal rules. Programs are analyzed at the function level, all inter-procedural analyses are done using the information contained in annotations.

In order to detect possible buffer overflows in C programs, the static analysis proposed in this paper contains two parts: the string buffer specifications and the annotated functions.

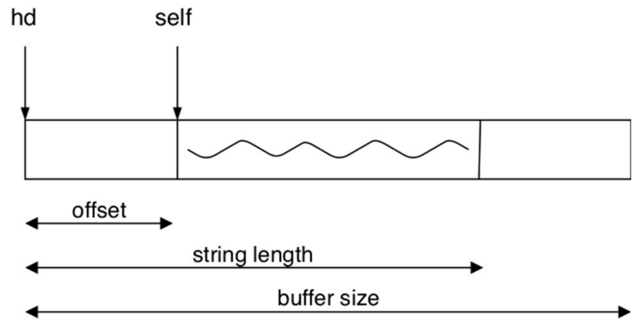

Fig. 1. Buffer Modeling in the Specifications

The string buffer memory area is defined through a set of properties that are encapsulated in a model understood by the prover, as in Fig. 1:

- A start address (*hd*). Once the memory is allocated, the start address has an invariant value. In a representational state, it could be perceived as the origin on an axis.
- A buffer size (*bsize*). Regardless of the part of the memory where the string resides, the buffer size has the same meaning and requirements. The buffer size is defined at allocation time and represents the size of the memory area that is reserved to hold data.
- A content size (*sl*). While the buffer size represents how much the system set aside for data, the content size defines the length of the actual content. When the system sets aside an area for a string buffer, it usually does not remove the pre-existing content, but allows it to stay there until it is overridden. Reading the pre-existing data is not an error that falls into the buffer overflow category, but it can be easily detected through this property in a static analyzer.
- A boolean value (*zeroflag*). It denotes the existence or non-existence of the null-terminating character. For a static analysis, it is not possible to detect what actually resides at a memory location during execution time. For setting the value of this property we based our analysis on some heuristics and only marked the existence of the character when we had certainty. For all the ambiguous situations, we set the value to False.

We defined a set of pre and post conditions for each vulnerable function exposed by libc. This mechanism simulates the memory behaviour inside the function and validates only the correct manipulations. The conditions are derived from the manual specifications.

## IV. Implementation Specifics

In this section we focus on pointer aliasing, and overlapping memory, double pointers, runtime values, use after free, loop heuristics, over-reads and under-read.

### *A. Pointer Aliasing*

The topic of pointer aliasing is widely debated in the context of a static analysis. It is hard to decide whether two variables point to the same location before programs's execution, thus many memory verifiers are nor sound or complete. In this regard, HIP/SLEEK offers good support[1].

With this support we managed to prove that the buffer overflow analysis provides good results also in the case of pointer aliasing. We applied the separation logic to declare two disjoint string memory areas:

```
    src::strbuf<hd_s, bsize_s, sl_s,
zeroflag_s>  *  dest::strbuf<hd_d,
bsize_d, sl_d, zeroflaf_d>
```

The predicate is only validated in case the two buffers were previously unified with separate *strbuf* data structures. Otherwise, the condition will result in failure. The following is one code example where the analyzer identifies that both *s* and *p* point to the same memory location and thus the verification will be successful. After the first *strcpy* call, the analyzer will acknowledge that both *p* and *s* point to a memory location that contains the string "smth".

```
char *s, *p, *t;
s = malloc(8);
t = malloc(8);
p = s;
strcpy(p, "smth");
strcpy(t, s);
```

It might be misleading to believe that the verification validates the previous piece of code because *s* points to a valid memory area since it was successfully allocated after the *malloc* call. In this case, it is important to remember that in our analyzer we consider that the result of the *malloc* function is a string buffer that is not null-terminated. In order to give more evidence, let's look at the following example:

```
char *s, *p, *t;
s = malloc(8)
t = malloc(8);
p = s;
```

```
strcpy(t, p);
```

The result outputted by the analyzer is the following:

```
Proving precondition in method
my_strcpy$char_star~char_star Failed.

(must) cause: !(zeroflag_s)
|- zeroflag_s.
```

The output states that it unifies on a source string that is not null-terminated *(!zeroflag_s)* and tries to prove the opposite (*zeroflag_s*). The *must* keyword that stands before the cause of the failure emphasizes that the prover could not entail the conditions. This is an important identifier of the result's reliability. It states that there must be a problem.

On the other side, the prover could result in a state where it *may_fail*. This could be a marker of a false alarm. An example where the entailment would result in a *may_fail* is the following:

```
y::strbuf<x,size,sl,true> -> sl < size-
10
```

Here, it unifies the *y* variable with a string buffer and tries to prove that the content size is less than (size-10). In this state, the prover knows that both *sl* and *size* validate the strbuf conditions, but that is not enough to conclude that *sl < size-10*. On the other side, it also cannot conclude the opposite either so it returns in a *may_fail* state. It means that it could fail or it could succeed but the information is not sufficient to have a strong conclusion.

### B. Overlapping Memory

Similar to the pointer aliasing case, in a static analysis it is difficult to make decisions whether two memory locations have overlapping fragments. It is particularly relevant for the overflow detection because in most of the string manipulation functions in libc, the behavior is undefined for overlapping strings.

For example, in the case of a copy function, the result depends mostly on the implementation of the system, whether it copies left-to-right or in the reversed direction.

Our buffer overflow verifier gives good results on detecting the overlapping memory using the separation logic.

In the following example the preconditions for the strcpy function would fail because the prover can identify that both the source and the destination reside in the same memory area. The result is guaranteed to be valid, because the offsetting function (src+4) will only succeed in case the offset value is within the range of the buffer size. Same condition is also validated in the *strbuf* data structure. This sustains the fact that the strcpy preconditions can only reason on the fact that the two buffers have a similar start address.

```
char *str = "string"
strcpy(str, str+4);
```

Same reasoning applies to the following case. The tool properly detects the error of the overlapping memories:

```
char *str = "string"
char *dest = str;
strcpy(dest, str);
```

### C. Use After Free

Because of the promising results, we expanded our goal and tried to detect not only buffer overflows, but also adjacent memory vulnerabilities.

Real world use after free vulnerabilities can be complicated. The allocation, propagation, free and dereference operation could all be located in separate functions and modules. This is a scenario where the HIP/SLEEK tool would fail to warn the user, but at a function level the results are promising for string memory areas.

We annotated the *free* function based on the following logic: a valid free requires that the parameter unifies on the strbuf data structure and ensures that in the postconditions, the string will no longer have the unification valid. This simple logic allowed us to have good results on the following input:

```
char *s, *a;
s = malloc(8);
a = s;
free(s);
free(a);
```

### D. Loop Unfolding

Statements involving control flow such as while/for loops and if statements, require more complex analysis than simple statement lists. All expressions are treated as separate functions so they require to be understood by the HIP/SLEEK logic. In case of a simple if statement, the tool infers the annotations and thus it is verified at the level of the function where is declared.

On the other side, the *for* and *while* loops need to be annotated in order to be verified. The prover treats the loop as a separate function and does the entailment independent of the outer function.

```
char *str = malloc(8);
for (int i = 0; i < 8; i++)
        str[i] = 'a';
```

The previous code overflows the buffer. This is a common error known as off-by-one where the null-terminator is overridden.The verifier successfully detects the overflow in the previous case and warns the user.

## V. Results and Evaluation

The verifier proved to have many great results in the buffer overflow detection. While there are still many things to improve, the tool proves that the exploration ground is favorable. This is a stepping stone towards the greater goal, a fully automated buffer overflow detector.

It should be clear by now that the proving tool we used offers support for a function level verification, so in the next paragraphs I will only focus on test cases where the strings are declared, initialized and manipulated inside the body of a single function.

We targeted test cases from three different categories as follows:

- A set of 50 C source files with buffer overflow vulnerabilities. I manually wrote them with the goal of covering as many corner cases as possible. I split them in three categories:
    - Possible overflow errors. Here I gathered all the test cases where the error is not guaranteed to happen, but it could in some cases. For example, from a user input or a allocated memory area where the existence of a null-terminator is not explicitly marked in the code.
    - Buffer overflow errors. I wrote a set of cases where the overflow is imminent. Some examples are: copy some content into a buffer where it does not fit or the result of two strings concatenation that is bigger than the destination buffer.
    - No overflows. Here I focused only on test cases where the prover has to return success. This proved that the verifier not only throws warnings for any string function call, but can properly entail the conditions on both safe and unsafe code.
- A set of inputs gathered from security competitions (CTFs). We have a set 10 cases proposed for buffer overflow exploits.
- Code used in nowadays programs that has known vulnerabilities. The testbed includes Wilander's tests [7] and open source programs like crashmail-1.6 and mawk-1.3.3.

Table 1 presents an overview of how the verifier behaves in certain situations:

TABLE I. Overview of Results

| | |
|---|---|
| heap overflow | good detection |
| stack overflos | good detection |
| global overflows | some cases |
| pointer aliasing | good detection |
| overlapping memory | good detection |
| use after free | good detection |
| function pointers | ignored |
| double pointers | good detection on strings |
| loop unfolding | source code needs to be instrumented |
| runtime values | generate many false alarms |
| inter-procedural verification | not supported |

Results on each test set and what cases they cover:

- On the set of 50 buffer overflow scenarios we had a high rate of detection. In 44 out of the 50, the errors were correctly identified as and reported to the user. The 6 cases generated false alarms. The verification failed because of the lack of tight specifications that could prove the correctness. All errors were marked as *fail_may*. It could be a good indicator for the user that the warning is an false alarms. While a *fail_must* probably comes from a real security vulnerability, the *fail_may* can indicate an false positive. This narrows down the list of errors that has to be manually verified by the user.

- The source code gathered from CTF competitions had good results too. Most of the security vulnerabilities could be exploited through command line arguments or user inputs. All the alarms are marked as *fail_may* which is accurate. It is not a given that all users are attackers. At the same time, the verifier reports correctly all the cases where an overflow is likely to happen.
- The open source code was trickier to test because the logic does not include only string function calls, but also logic on other data structures that are not translated yet into the verifier. The input had to be manually parsed in order to remove unnecessary code and to annotate the loops. The results are quite encouraging. The verifier generated a series of warnings that indicate some vulnerabilities. Most come from *strcpy* function calls that can indeed overflow the destination buffer because there is no bound check in the code. One down side was the loop annotation. The conditions were hard to translate into the HIP/SLEEK logic and thus some were ignored. As an example, after running the verifier on a function input (*filter_evalfunc*) copied from the Crashmail source code, there were two *accurate* overflow vulnerabilities and no false alarms.

We evaluated the time taken using HIP/SLEEK to verify the C programs in our test bed. Times were in the order of seconds making our approach efficient for verification of C programs.

## VI. Conclusion

In this paper we presented an automated approach for verifying buffer overflow vulnerabilities in C programs. The prototype proved to have motivating results and encourage us to look for further improvements. This is one step towards the ideal state of a fully automated static analyzer capable of detecting all security vulnerabilities.

Compared with other static error detector, this approach has made the following advances: (1) it is able to treat memory areas as separate and independent locations, thus it offers good precision in handling pointer aliases, (2) it aims for a static verification of string manipulating functions from libc, but comes with a series of other annotated functions that improve the precision of the results(3) it diagnoses not only buffer overflows, but also other security vulnerabilities, such as use-after-free, double free, under/over-reads, (4) it provides a set of built-in predicates, but allows arbitrary user-specified (inductive) predicates that improve the result's accuracy based on user's particular case, (5) it generates precise error messages about the source of the failure.

Our current heuristics strike a balance between (i) a low rate of false alarms by applying tight conditions, and (ii) a fast and accurate function level buffer overflow verification.

While the results are favorable and the prototype properly diagnosed the test cases, we would like to look more into two areas. One is an automated verifier that performs inter-procedural checks which would increase the reliability of the results. The other one is an automated verifier complemented by an automated inference system. This would allow us to skip the translation step and run the tool directly on the source code. At the same time, the tool would be capable to audit substantial sizable source code that is used in today's programs.

## Acknowledgment

This work was supported by a grant of Romanian Ministry of Research and Innovation, CCCDI - UEFISCDI, project number PN-III-P1-1.2-PCCDI-2017-0272 / 17PCCDI-2018, within PNCDI III.

## References

[1] W.-N. Chin, C. David, H. H. Nguyena, and S. Qinb. Automated verification of shape, size and bag properties via user-defined predicates in separation logic. Under Consideration by Science of Computer Programming, 2009.

[2] N. Dor, M. Rodeh, and M. Sagiv. Cleanness checking of string manipulations in C programs via integer analysis. The 8th International Static Analysis Symposium, 2001.

[3] D. Larochelle and D. Evans. Statically Detecting Likely Buffer Overflow Vulnerabilities . In Proceedings of the 10th USENIX Security Symposium, pages 177–190. Network and Distributed System Security Symposium, 2001.

[4] W.-N. Lee, C. Song, H. H. Byoungyoung, and S. Tielei. Preventing Use-after-free with Dangling Pointers Nullification.

[5] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken. A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities. Network and Distributed System Security Symposium, February 2000.

[6] HIP/SLEEK. http://loris-5.d2.comp.nus.edu.sg/hip/. Accessed July 2018

[7] Wilander buffer overflow test suite. http://www.cs.virginia.edu/mc2zk/security_test_- suites/, Accessed July 2018