Program Analysis **November 22, 2015**

Lecture 5: CSSV - Detecting Buffer Overruns in C Programs

Lecturer: Mooly Sagiv *Scribe: Misha Seltzer*

5.1 Introduction

This lecture is based on paper by Nurit Dor, Micahael Rodeh and Mooly Sagiv.

The research was done as part of the Daedalus project, which combines academic research with real world applications in the industry. In the case of this paper, work was done in association with the Aerobus company. The paper was presented in the Programming Language Design and Implementation (PLDI) conference in 2003.

This paper is a good example of real application of what we've learnt until now.

5.2 Motivation

Even in a small C program, which does string manipulation, can easily contain several bugs. Such bugs may sometimes be hard to find, and create errors which won't crash the program, rather than make it work unexpectedly (for example, return the wrong output). Moreover, those bugs may create more hacking attack surface for the system.

5.2.1 Example

```
/* from web2c [strpascal.c] */
void foo(char *s) {
    while (*s != '')      /* Possible null dereference */
       s++;      /* Possible out of bounds pointer arithmetic */
    *s = 0;      /* Possible out of bounds update */
}
```

5.2.2 Current state

The general belief is that those bugs are fairly common. CERT advisory (United States Computer Emergency Readiness Team) report on many security holes that result from buffer overflow (i.e., updates beyond the bounds of a buffer).

5.2.3 FUZZ testing

Fuzz testing is a method for testing programs by supplying them with many different, somewhat unexpected, inputs.

A good FUZZ tester would have a basic understanding of the underlying function, and will supply it with different "kinds" of inputs. For example, for a char* input, it might give a NULL value, a very long string, and a string that is not NULL-terminated.

There are many FUZZ-testing systems, like Microsoft's SAGE: Whitebox Fuzzing for Security Testing system.

5.3 Goals

The goal of CSSV (C String Static Verifier) is to be an efficient, conservative static checking algorithm that:

- Verifies the absence of buffer overflows (not just finding bugs).
- Supports all C constructs (Pointer arithmetic, casting, dynamic memory, etc.)
- Can run on real programs
- Minimizes the number of false alarms

5.4 Examples

5.4.1 Complicated example

Let's inspect the following code, taken from the web2c project (implementation of TeX):

```
/* from web2c [fixwrites.c] */
#define BUFSIZ 1024
char buf[BUFSIZ];
char *insert_long(char *cp) {

    char temp[BUFSIZ];
    ...
    for (i = 0; &buf[i] < cp ; ++i)
        temp[i] = buf[i];
    strcpy(&temp[i],"(long)");      /* (1) */
    strcpy(&temp[i+6],cp);      /* (2) */
    strcpu(buf, temp);
    return cp + 6;
```

This code adds the string "$(long)$" at the $offset(cp)$ position of the buf buffer.

The possible errors that may happen here are:

- In (1), we have a potential cleanness violation for $7 + offset(cp) \geq BUFSIZE$.
- In (2), we have a potential cleanness violation for $7 + offset(cp) + len(cp) \geq BUFSIZE$.

5.4.2 Manual example

Verifying the absence of buffer overflows is a non-trivial task. In many cases, when manual assessment is required, it is easier done in reverse procedure order.

Let's examine the following program:

```
void safe_cat(char *dst, int size, char *src) {
    if (size > strlen(src) + strlen(dst)) {      /* (1) */
        dst = dst + strlen(dst);      /* (2) */
        strcpy(dst, src);      /* (3) */
    }
}
```

Let's inspect the code above going backwards

- line 3: string(src) $\wedge$ alloc(dst) $>$ len(src)
- line 2: string(src) $\wedge$ string(dst) $\wedge$ alloc(dst+len(dst)) $>$ len(src)
- line 1: string(src) $\wedge$ string(dst) $\wedge$ (size $>$ len(src)+len(dst)) $\Rightarrow$ alloc(dst+len(dst)) $>$ len(src))

(We define string(s) to be true for s being a NULL-terminated char*)

5.5 Real programs

5.5.1 Can this be done?

- We can use Polyhedra to find out complex linear relationships
- We can use Points-to-analysis to validate pointer arithmetic
- We can use widening to figure out loops
- We can request procedure contracts to figure out procedures

5.5.2 The CSSV

The presented system detects the following string violations:

- Buffer overflow (update beyond bounds)
- Unsafe pointer arithmetic
- Reference beyond null termination (Although not necessarily a bug, might point to a problematic design)
- Unsafe library/procedure call

It handles full C, including muli-level pointers, pointer arithmetic, structures, casting, etc.
And it applied to real programs (both from public domain, and from Airbus C codebase).

5.5.3 Operation semantics

The systems stores a "shadow memory", which, for each relevant memory address, holds the size of the allocated memory, and the address of the beginning of that allocated block.

Furthermore, CSSV's abstraction ignores exact locations in memory, and only tracks base addresses, sizes and pointers from one base address to another (with offset).

In that abstraction, CSSV assumes there's only one continuous heap memory chunk "$heap_1$" of size m. If we have pointers p_1 (at $0x480590$) and p_2 (at $0x480580$) both point to the heap, where $p_1 = heap_1$ and $p_2 = p_1 + 8$, we'll track that p_1 points to $heap_1$ and that p_2 points to $heap_1 + 8$ (without calculating the actual address of $heap_1 + 8$.

With that, pointer validation becomes a simple arithmetic validation problem. For example, with the pointer arithmetic statement $p_2 = p_1 + i$, we need to validate $*p_1.size \geq p_1.offset + i$, and with the pointer dereference statement $p_3 = *p_2$, we need to validate $*p_2.size \geq p_2.offset$.

5.5.4 The null-terminated byte

As pointed out earlier, we assume that it is illegal to write (or read) after the null-terminated byte (\0). With this assumption, it is enough to store (for each char*) only the first null-terminated byte (or store that it doesn't exist).

It's important to know when the byte does not exist as well, since it is imperative to knowing legality of calls to some functions, like "strcpy" or "strlen".

5.5.5 Procedure calls - contracts

We require the C code procedures to be annotated with standard contracts. Those contracts define for each procedure what are its requirements, what it modifies, and what it ensures.

The advantages of such contracts are:

- Modular analysis in case of missing code
- Lets us run more expensive analysis by siloing out code that needs not to be checked.
- Improves the precision of the analysis
- Can check additional properties (beyond ANSI-C)
- Detects errors in point of logical error (Tell the programmer that the call is done with wrong parameters, rather than fail some condition inside the procedure).

Example of such contract:

```
char* strcpy(char* dst, char *src);
/* Requires    string(src) ∧ alloc(dst) > len(src)
   Mod         len(dst), is_nullt(dst)
   Ensures     len(dst) == pre@len(src) ∧ return == pre@dst */
```

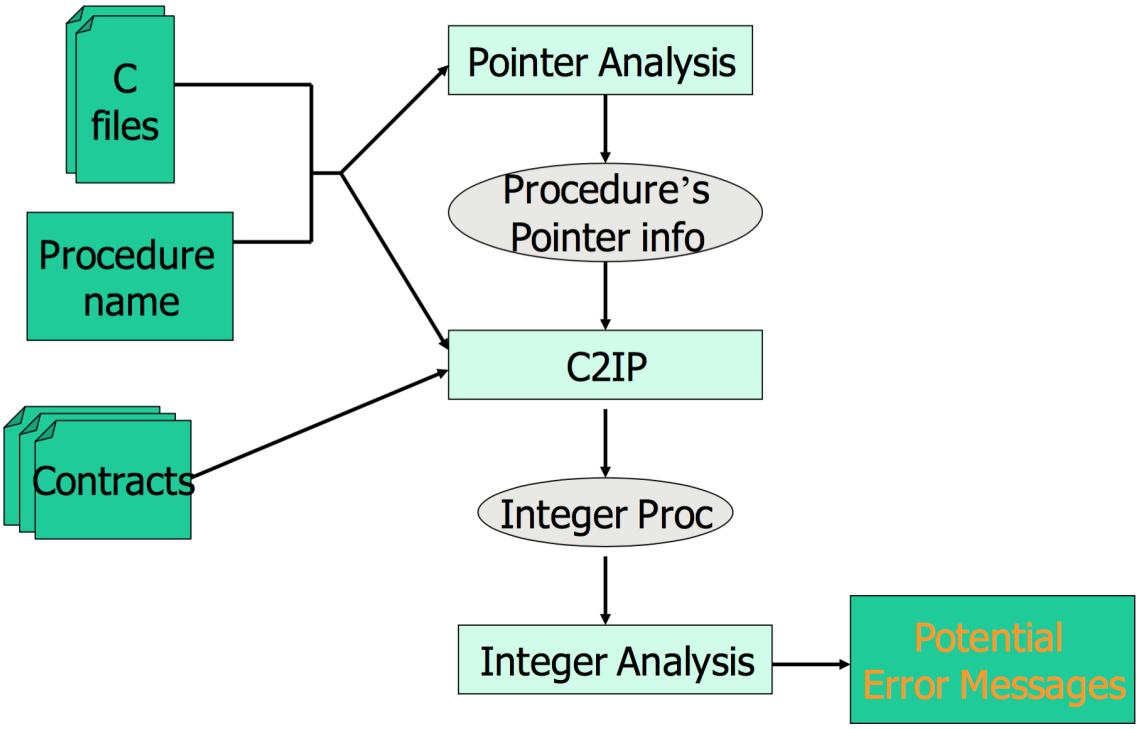
The mod label is required to eliminate the need for defining in precondition things that happen to parameters outside of the scope of the function.

Coming back to the "Complicated example" at 5.4.1, the contract should be as following:

```
char insert_long(char *cp);
/* Requires    string(cp) ∧ buf ≤ cp < buf + BUFSIZE
   Mod         cp.len
   Ensures     len(cp) == pre@len(cp) + 6 ∧ return_value == cp + 6 */
```

5.5.6 Technical overview

CSSV first takes ALL C code, and runs full points-to analysis to produce procedure's pointer information. That information together with the contracts is fed into the C2IP to transform the problem into an integer program. From that point, and integer analysis is ran to produce relevant errors.



5.5.7 Results

The results of the CSSV seem to be encouraging, with high recall and a very low count of false alerts, although for a bad penalty in memory usage (most used for the Polyhedra).