Enforcing Security in Programming Languages using Separation Logic

Horia V. Corcalciuc

University of Birmingham

March 14, 2008

- Increasing number of attacks on **unsafe code** usually based on the lack of **logical reasoning** at the programming stage.
- The lack of a logical implementation to reason about issues such as concurrency in **low level languages**.
- Two dead projects: **Vault** and **Cyclone**. The former is developed by Microsoft but ceased to be updated in 2004 and not even supporting threads while the latter seems abandoned.
- No complete dialect of C implementing logical reasoning such as **Hoare** and **separation logic**.

What we can do

- Reason about exploits using logic and try to eliminate them using **static** and **runtime checks**.
- Implement a safe dialect of C that will **refuse to compile insecure code**.
- Port libraries to make code compiled against them be **partially secure**.
- Implement Hore-logic pre- and postconditions at **language level** allowing the programmer to implement his own checks.
- Make non thread-safe functions thread safe and non reentry safe functions reentry safe.

Hoare and Separation Logic

- We use Hoare logic to reason about program flow.

Example

$$\{precondition\}\ command_1\ \{postcondition_1\}$$
$$\{postcondition_1\}\ command_2\ \{postcondition_2\}$$

- We use separation logic to reason about memory.

Example

$\{emp\}$ The heap is empty.

$\{x \mapsto x'\}$ The heap contains one cell at address x with contents x'.

$\{P * Q\}$ The heap can be split in two disjoint cells. P holds for one, and Q for the other.

$\{P \mathrel{-\!\!*} Q\}$ If the heap is extended by a disjoint part in which P holds, then Q holds for the extended heap.

What we try to prevent

- Overflows (buffer, heap stack, integer etc...)
- Format string attacks
- Code injections
- Race Conditions

What we can prevent

- Overflows (buffer, heap stack, integer etc...)

Defense

Bounds checking, regions and upcasting.

- Format string attacks
- Code injections
- Race Conditions

Defense

Mutex locks and re-entrant functions.

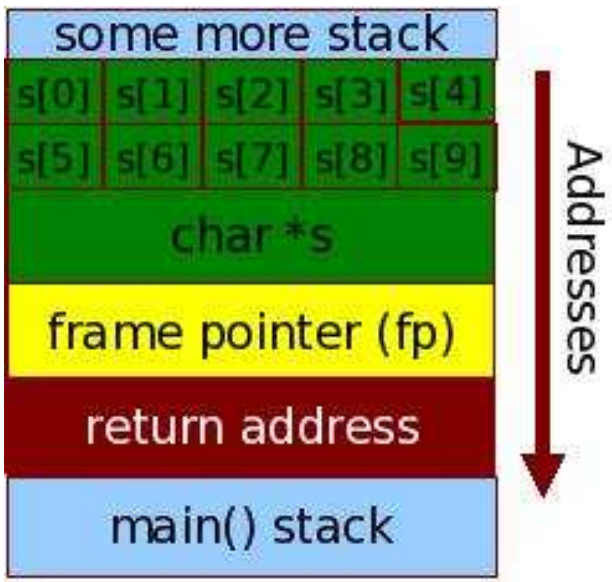
A buffer/stack overflow example

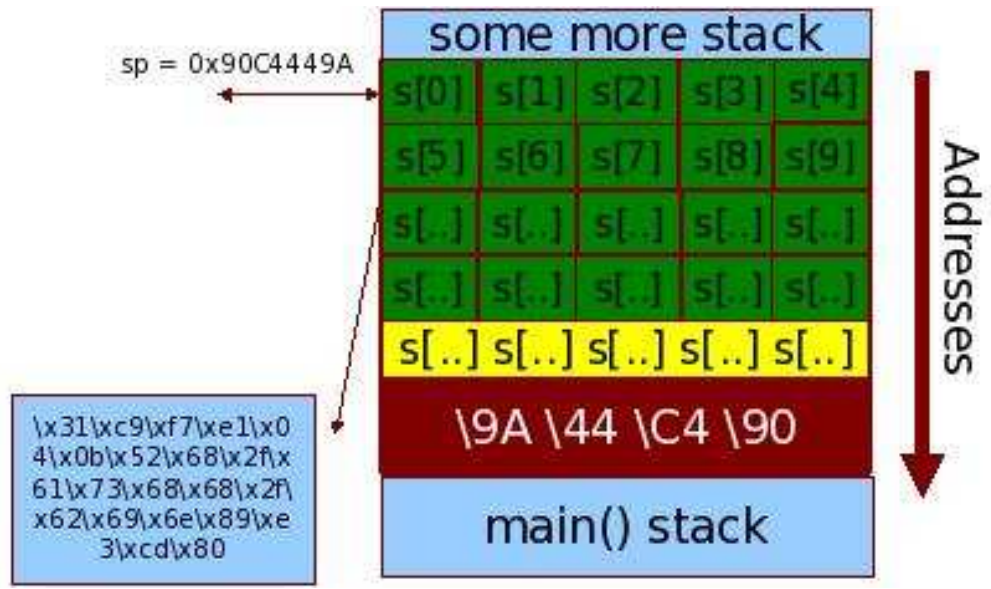
```
void f(char *data) {
  ...
  char s[10];

  /* No bounds check done */
  strcat(s, data);
  ...
}

void main(int argc, char **argv)
{
  f(argv[1]);
}
```

Buffer overflow diagram





Bounds checking for buffer overflows

- We model the allocation of an array in Hoare logic:

Allocation

$$\{emp\}\, x = new(k)\left\{\overset{k-1}{\underset{i=0}{*}}[x+i] \mapsto -\right\} \text{where } \overset{k-1}{\underset{i=0}{*}} x_i = x_0 * x_1 * \ldots * x_k$$

- Knowing the size of the allocated array we can reason on indexes whenever a pointer operation takes place:

Attempted assignment

$$\frac{\left\{\overset{k}{\underset{i=0}{*}} a+i \mapsto - \wedge j \leqslant k\right\}[a+j]:=m\left\{\overset{k}{\underset{i=0}{*}} a+i \mapsto -\right\}, \left\{\overset{k}{\underset{i=0}{*}} a+i \mapsto - \wedge j > k\right\} skip \left\{\overset{k}{\underset{i=0}{*}} a+i \mapsto -\right\}}{\left\{\overset{k}{\underset{i=0}{*}} a+i \mapsto -\right\} \text{if } j \leqslant k \text{ then } [a+j]:=m \text{ else skip fi} \left\{\overset{k}{\underset{i=0}{*}} a+i \mapsto -\right\}}$$

Integer overflow example

- Suppose that integers a and b hold the maximum possible value an integer of type “int“ can hold.
- The result will be undefined. It may either wrap around or just truncate at maximum value.
- In case “result“ would hold logical meaning or used to manage memory the results could be undefined.

```
...
int a, b, result;
result = a + b;
...
```

Upcasting for integer overflows

- Given two integers defined on 8 bits and a third one defined on 16 bits:

Precondition

$\{\exists a, b, result \in int8\_t \wedge result = a + b \wedge \exists temp \in int16\_t \wedge int8\_t \subset int16\_t\}$

- Upcast one member of the sum and store in the temporary variable:

Command

$temp = (int16\_t)a + b;$

- If it does overflow, don't proceed with the assignment, otherwise downcast and assign. We use the IF axiom to infer:

IF axiom, precondition, assignment and postcondition

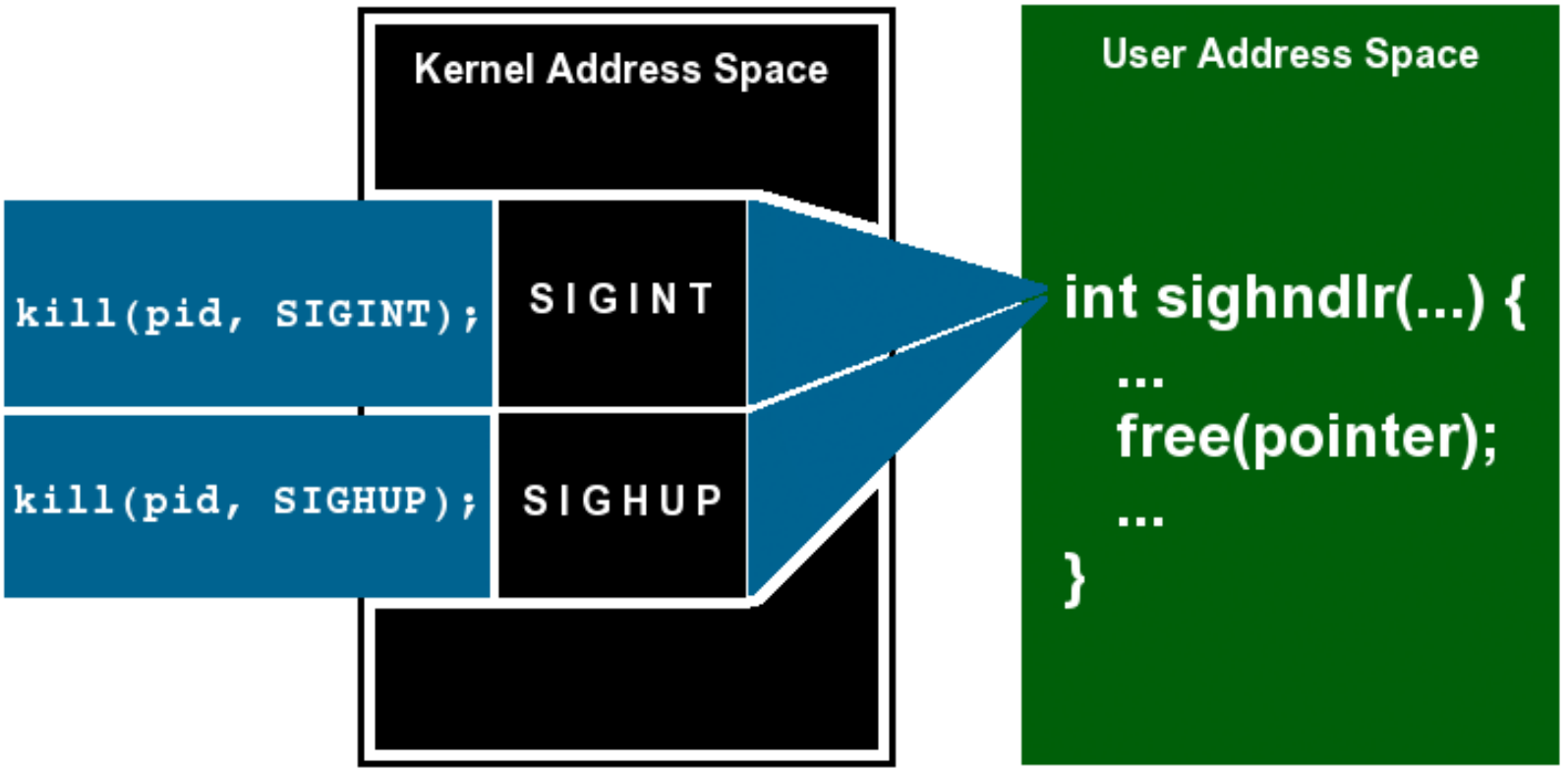
$\{temp = a + b\}$if $temp < max(int8\_t) \wedge temp > min(int8\_t)$ then
$result = (int8\_t)temp;$ else $error$ fi $\{result < max(int8\_t) \wedge result > min(int8\_t)\}$

A concurrency example

```
char *global_pointer;
...
void f() {
...
        free(global_pointer);
...
}

int main(void) {
...
        global_pointer = (char *) calloc(10, sizeof(char));
...
        signal(SIGHUP, f);
        signal(SIGTERM, f);
...
}
```

Buffer overflow diagram



Using mutex locks and flags

- We use mutex locks for thread safety and a flag for re-entry safety:

Mutex locks and flags

$\{\exists x.x \mapsto lk = 0 \wedge \exists flag.flag = 0 * Q\}$
pthread_mutex_lock(ptr);
$\{x \mapsto lk = TID \wedge flag = 0 * Q\}$
dispose(ptr);
$\{x \mapsto lk = TID \wedge flag = 0 * (ptr \mapsto \_) \mathrel{-\!\!*} Q\}$
[flag]=[flag]+1;
$\{x \mapsto lk = TID \wedge flag = 1 * Q_{rest}\}$
pthread_mutex_unlock(ptr);
$\{x \mapsto lk = 0 \wedge flag = 1 * Q_{rest}\}$

What we intend to do...

- Analyse and gather data about common exploits and research their cause.
- Introduce **programming features** instead of limiting the user but making sure they have a sound foundation in programming logic.
- Use **separation logic** and **Hoare logic** to describe events happening in a program to eliminate the possibility of any point of failure.
- Analyse code and decide what is secure and what not and point out the problem to the programmer.