HAIT: Heap Analyzer with Input Tracing

Andrea Atzeni
Andrea Marcelli
Francesco Muroni
Giovanni Squillero

Andrea Marcelli
Ph.D. Student at Politecnico di Torino
“Machine Learning for Security Applications”

Main focus on automated malware analysis, malware families identification and signature generation

Experience with application reversing (PE, Android), software exploitation and security challenges. Hacking IoT devices in the free time

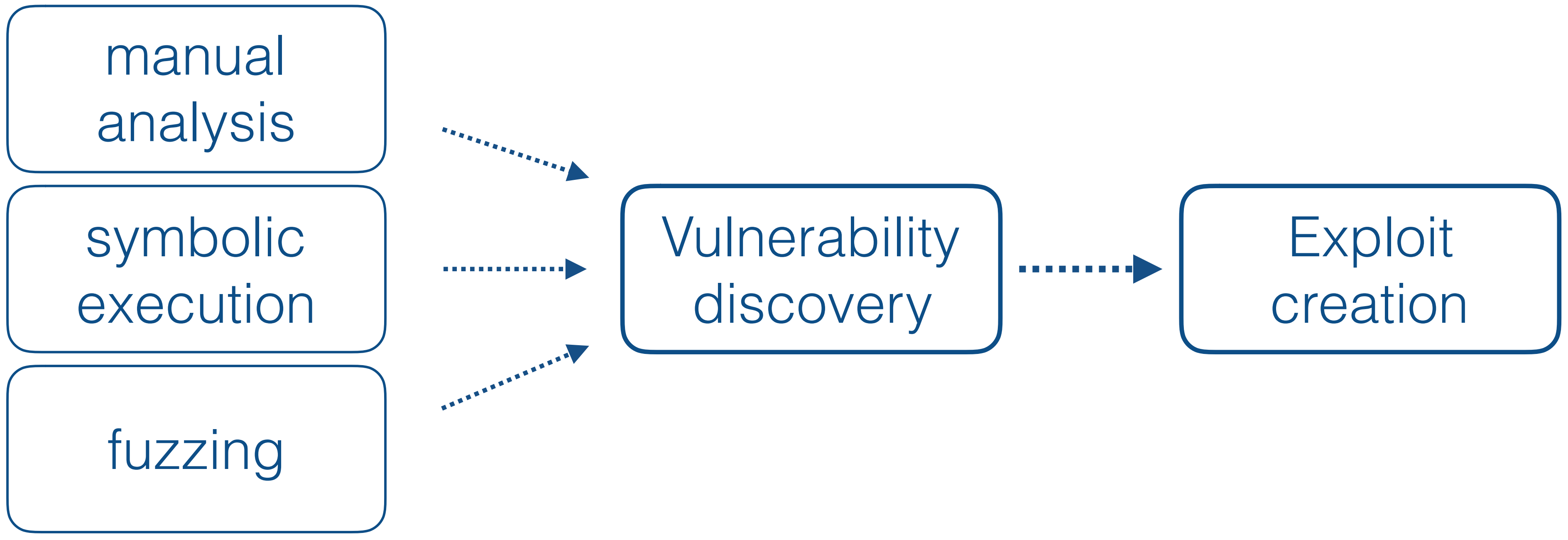
Goals

Assist security researcher in the gathering of essential information to exploit heap-based vulnerabilities

HAIT is a POC tool developed to assist the exploitation phase, but it can be used for training and didactics too.

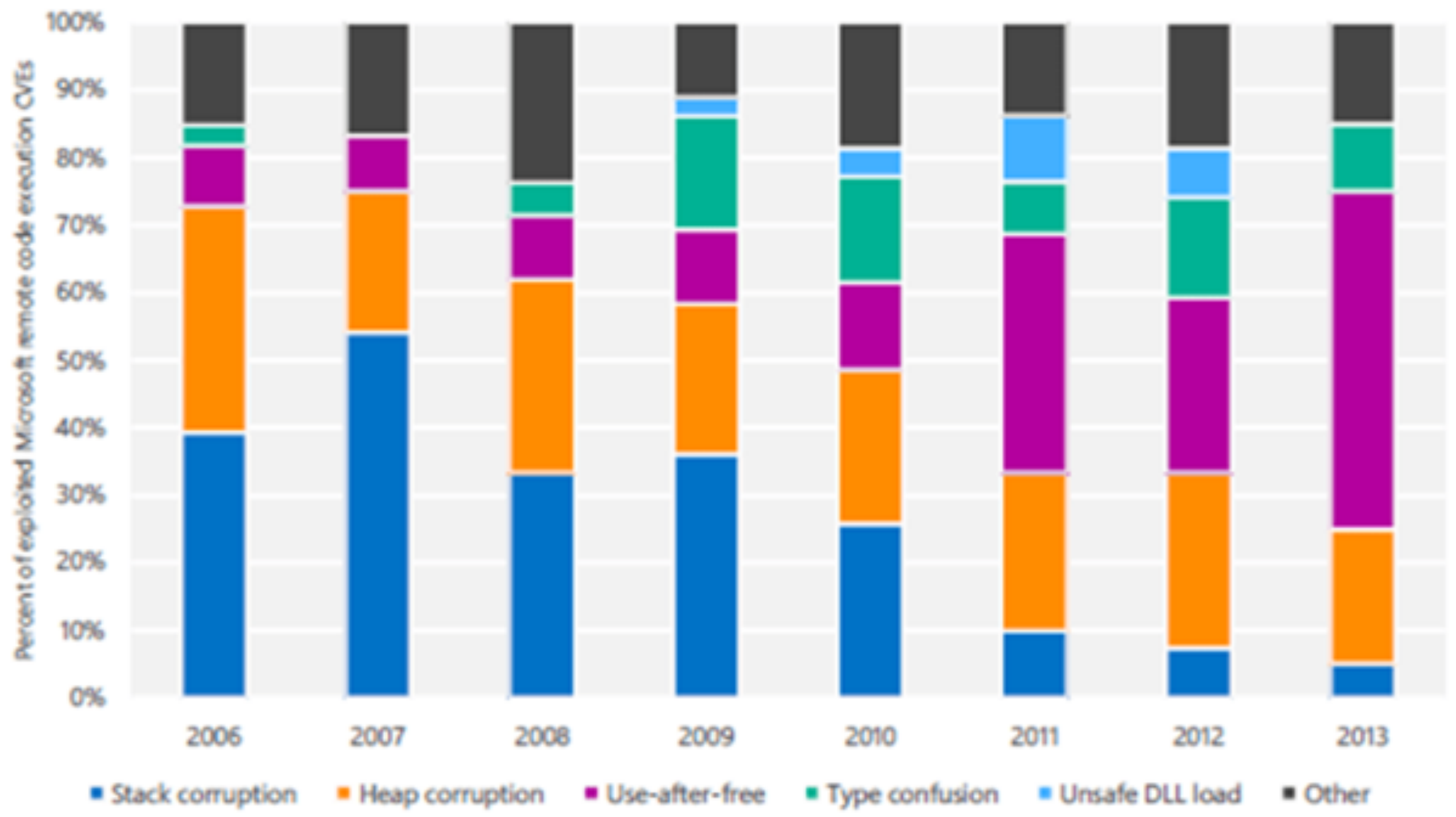
Exploit Development

An **exploit** is a combination of code and data that takes advanced of a flaw in a system to cause an unintended behavior (e.g., gaining illegitimate control)



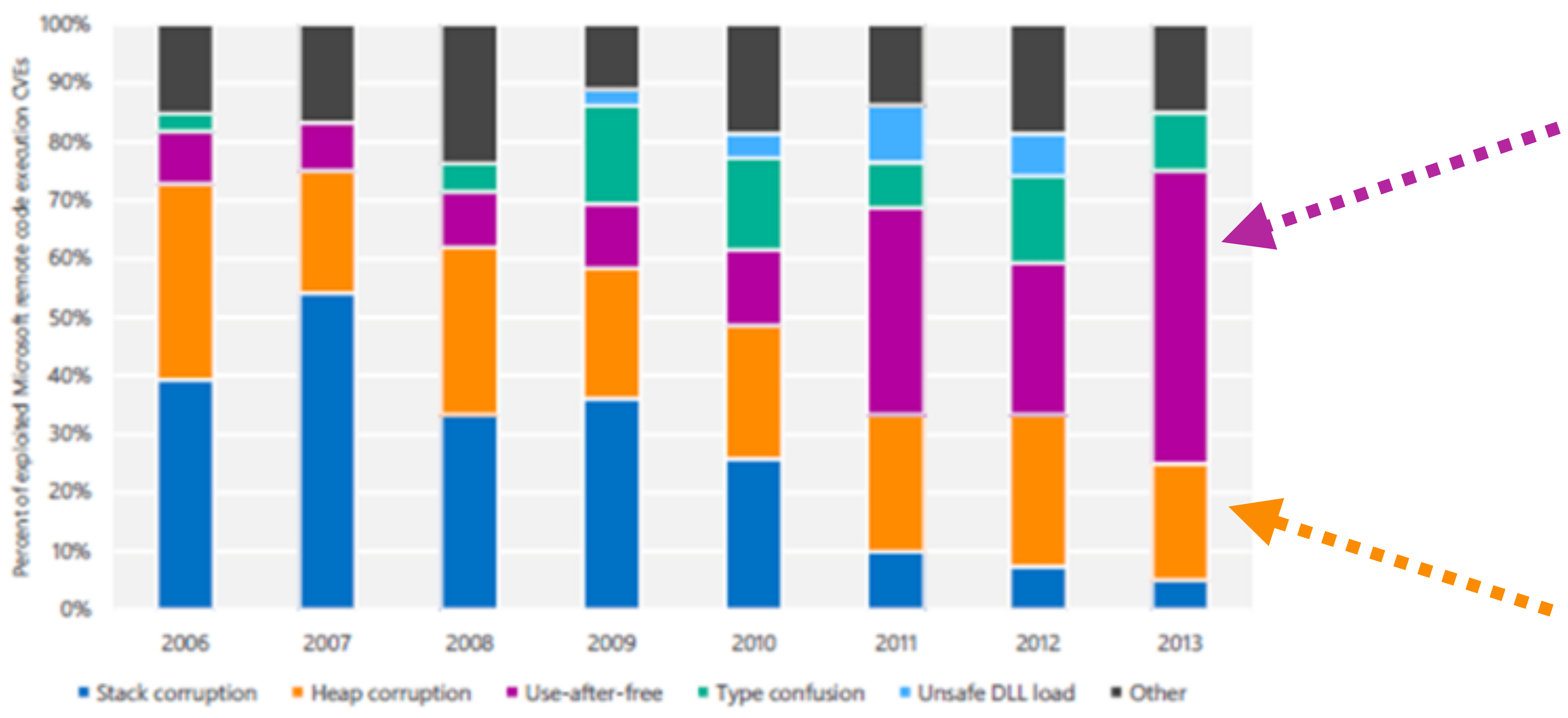
Causes of Remote Code Execution

Stack-based exploitation is becoming **rare**, most common attacks now rely on the heap



Causes of Remote Code Execution

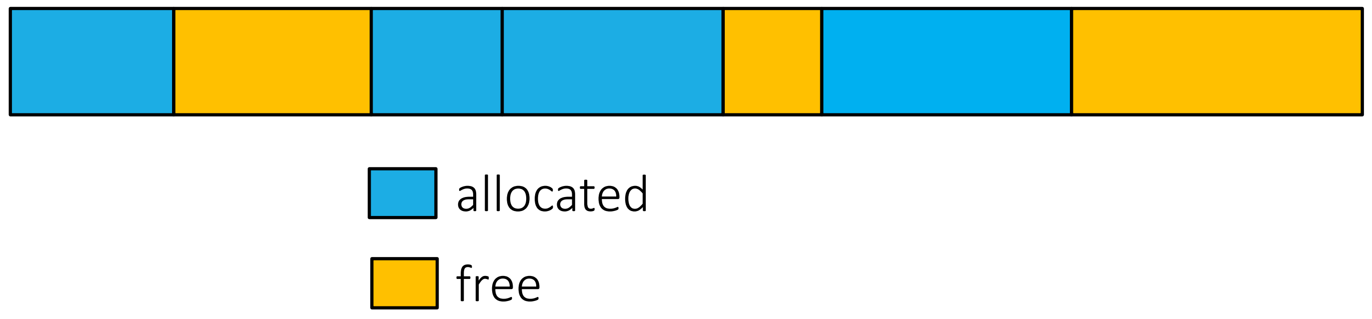
Stack-based exploitation is becoming **rare**, most common attacks now rely on the heap



HEAP

Data structure used to manage dynamically allocated memory

- **Allocation**: reserve a block of memory of the requested size
- **Free**: mark a previously allocated block as not used, available for future allocations



Problems of heap exploitation

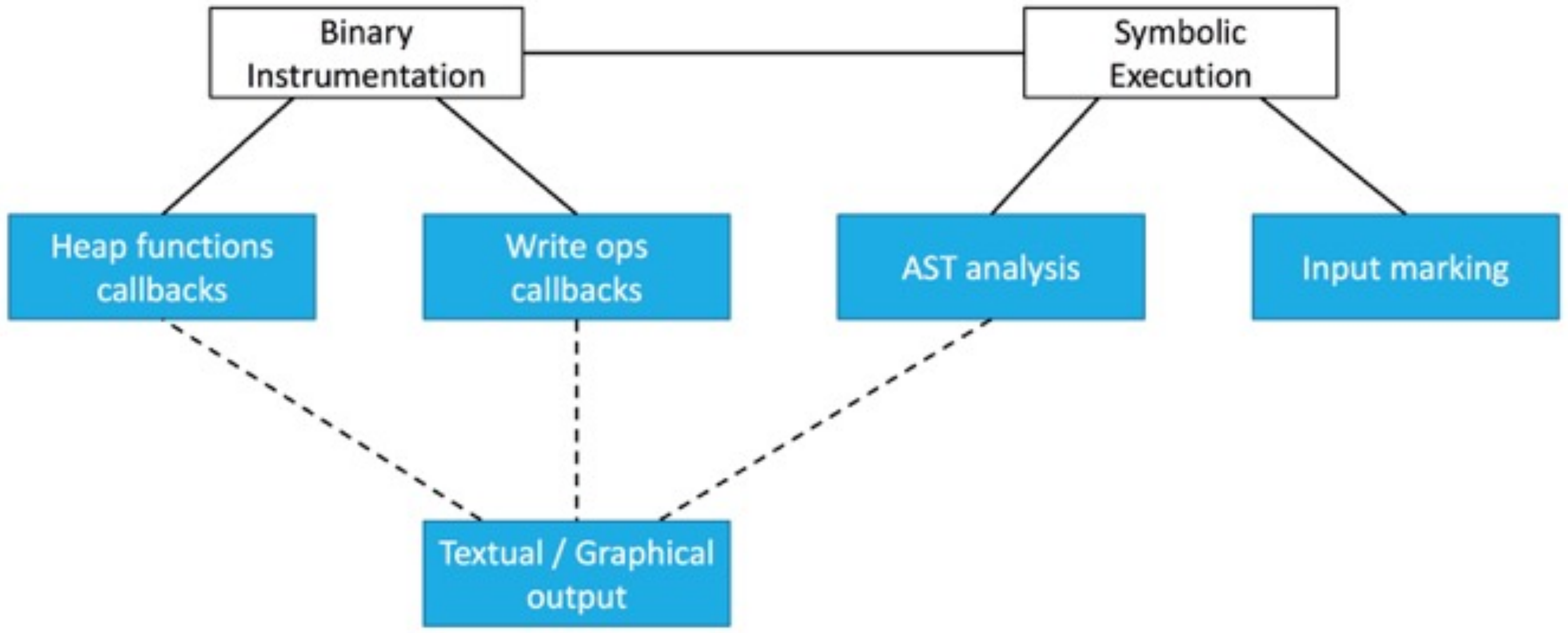
- Multiple complex data structures and functions involved
- Specific to an allocator
- Many different techniques, each requiring specific heap layout and sequence of operations

As the level of complexity of the target systems grow, **manual inspection** of memory to retrieve the necessary data becomes **unfeasible**

Our contributions

- **Identification** of the essential information for heap exploitation
- **Definition** of a methodology to automatically gather such information at runtime
- **Implementation** of the methodology in a tool, HAIT

HAIT Schema



Binary Instrumentation
Symbolic Execution
Heap functions callbacks
Write ops callbacks
AST analysis
Input marking
Textual / Graphical output

Which information?

Two categories of data are needed:

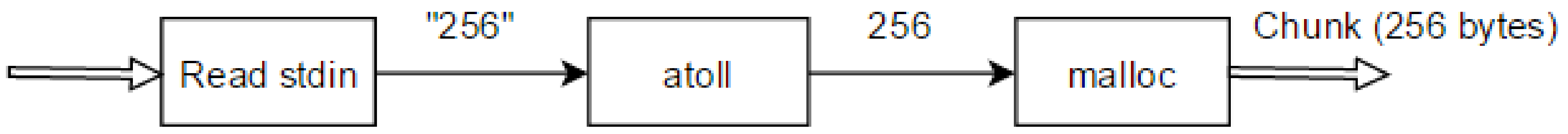
- Heap state at each point of the execution
- Influence of the program inputs on the heap state

Heap state

- Block location, size, allocation state, metadata
- Write operations that occur in the heap memory region, most of all if any metadata is overwritten

Program input

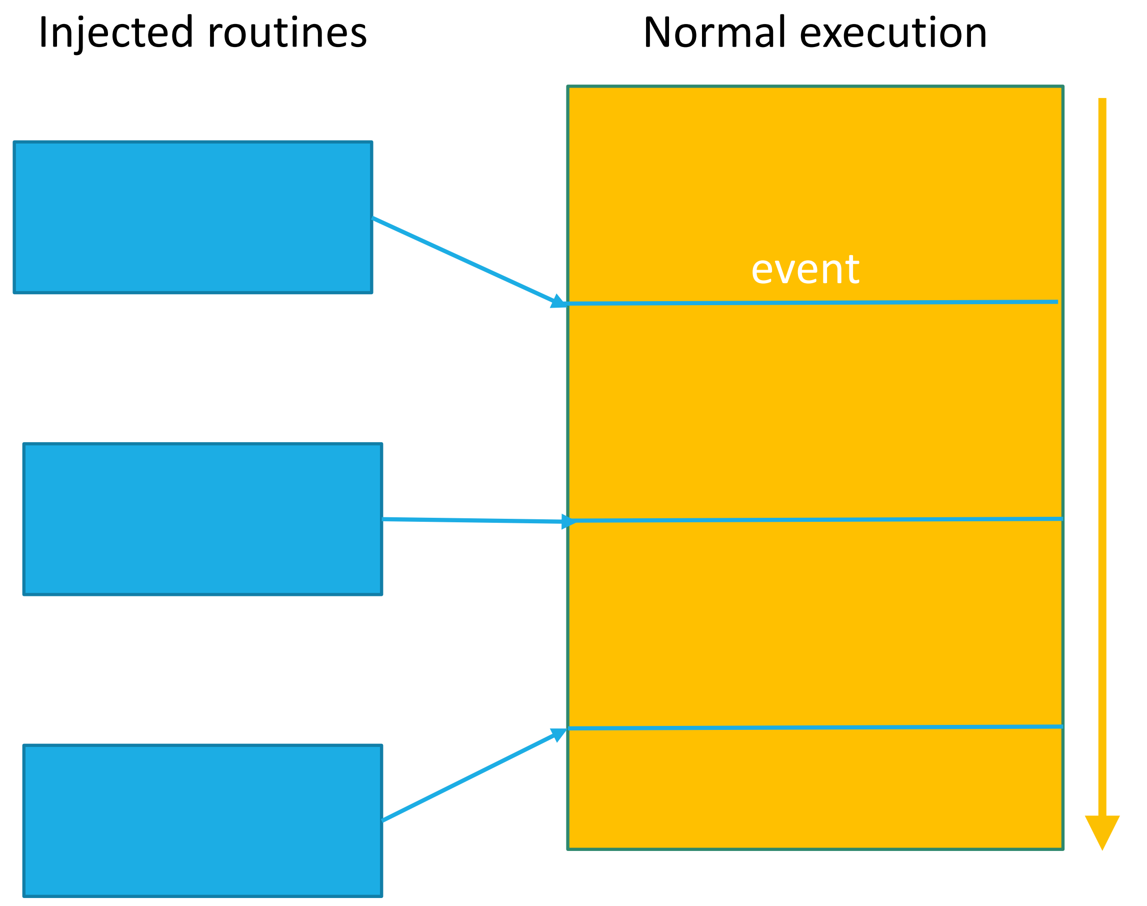
- Input comes from many sources: standard input, command-line arguments, environment variables, files, sockets.
- Input is stored as a sequence of bytes. Tracking the input means to track the use of those bytes.
- We are interested in the heap operations that depend on the input.



Binary instrumentation

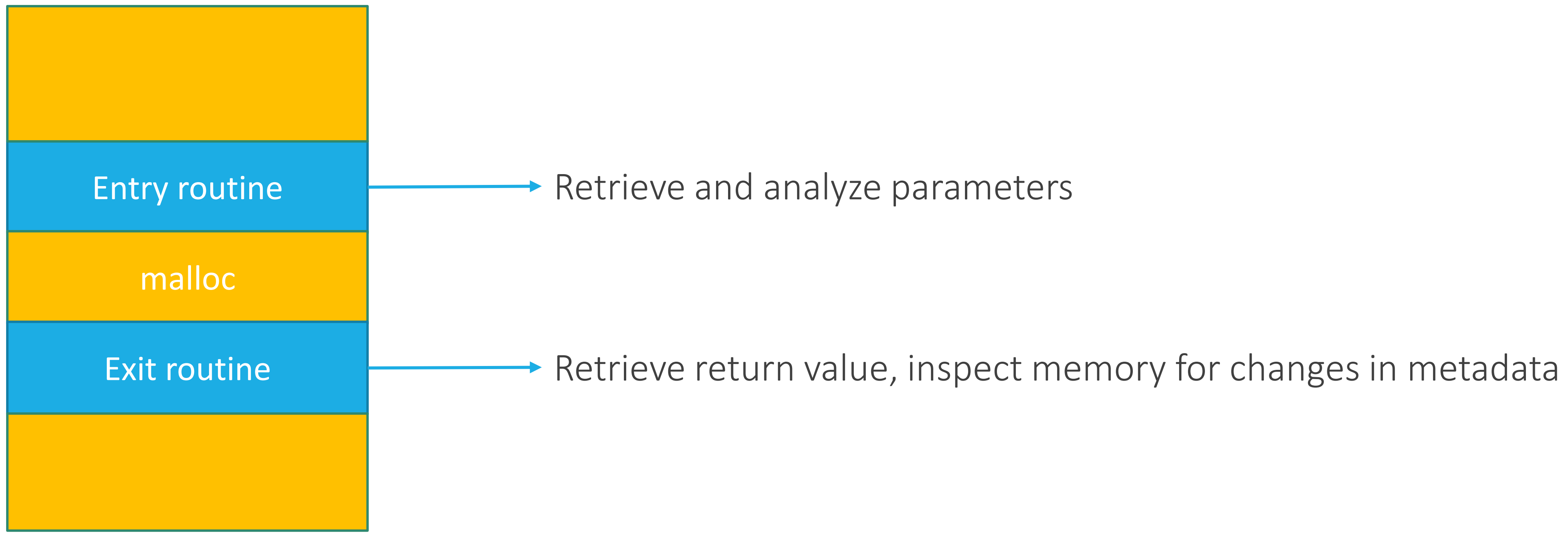
- Injection of extra code in the normal execution flow of a binary in the form of routines

- These routines are executed when specific events occur.



Binary instrumentation

To record calls to heap functions, we execute analysis routines before and after the function is executed.


Entry routine
malloc
Exit routine
Retrieve and analyze parameters
Retrieve return value, inspect memory for changes in metadata

Symbolic Execution

The symbolic execution engine translates instructions into symbolic expressions, which are associated to registers/ memory locations that they modify

add rax, rdx $\Longrightarrow$

	ID
RAX	Ref!41
RDX	Ref!39

	SymExpr
Ref!39	…
Ref!40	…
Ref!41	Ref!39 + Ref!40

Symbolic Execution

To trace the inputs we perform two actions:

- Mark memory locations with program inputs as symbolic variables
- When a heap function is called, retrieve the symbolic expressions associated to the parameters and analyze them to find references to symbolic variables

Developing a proof of concepts

Heap **A**nalyzer with **I**nput **T**racing: proof of concept implemented as a Python script on top of the Triton framework. Records all heap operations, highlighting those that depend on the user input.

- **Textual output**, providing all the details of each operation (allocation, deallocation, write)
- **HTML graphic** representation of the evolution of the heap during execution, with details of the current state

Target: Linux 64-bit executables (*ptmalloc* allocator)

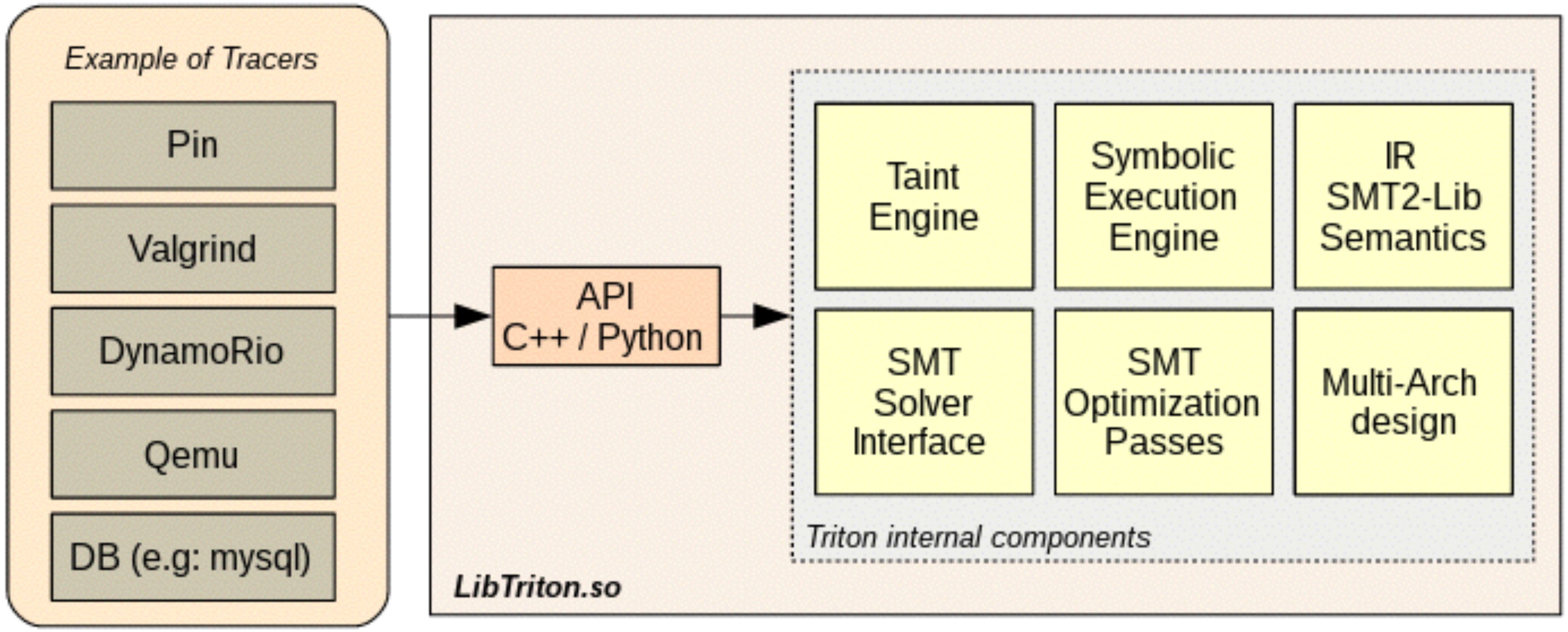
Triton*

- Modular framework for Dynamic Binary Analysis

*Florent Saudel and Jonathan Salwan, 2015

- We chose to use Pin** as dynamic binary instrumentation tool

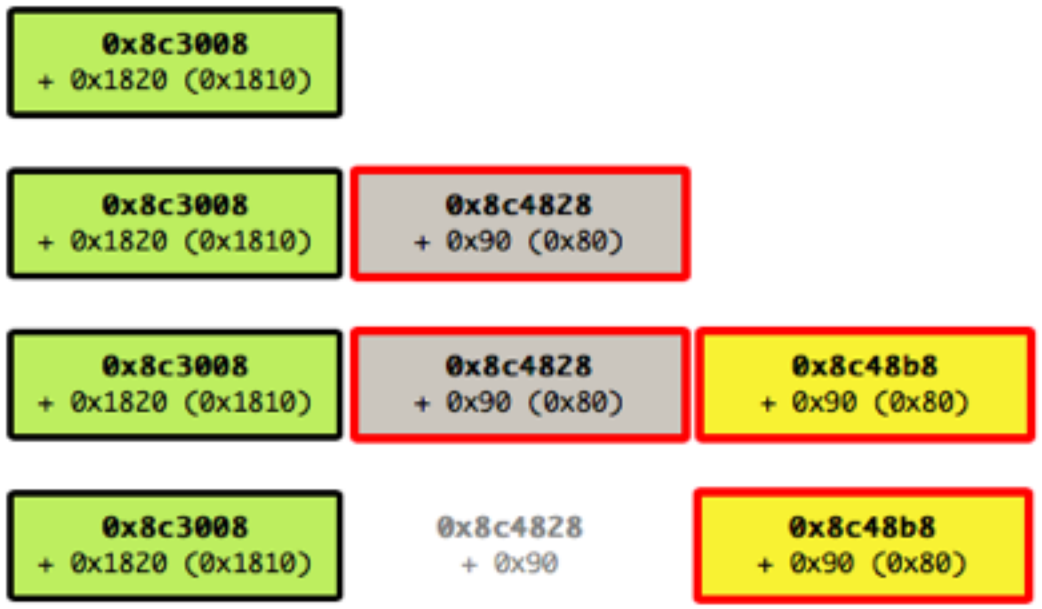
**Intel, 2012

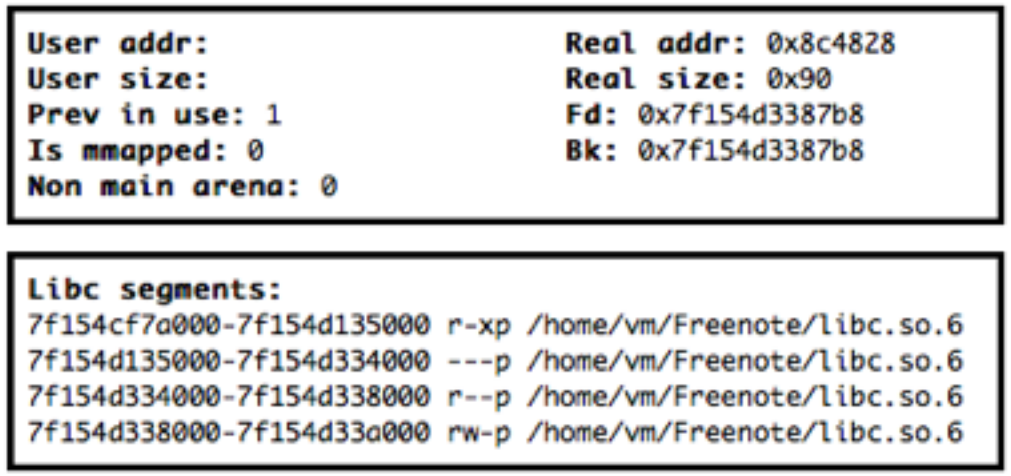


Textual output

```
Your choice : 2
Length of new note : 10

[*] Malloc -> addr = 0x8c4828, usize = 0x80 , rsize = 0x90
SymVar_0 read #1 byte 0 -> value='2' -> atoi
SymVar_2 read #3 byte 0 -> value='1' -> atoi
SymVar_3 read #4 byte 0 -> value='0' -> atoi

[*] Continued write operation in block 0x8c3008, starting at 8c3018
(current size 0x18)

[*] Write operation in block 0x8c3008, at 8c3010 (size 0x8) value = 0x1
```

Graphical representation

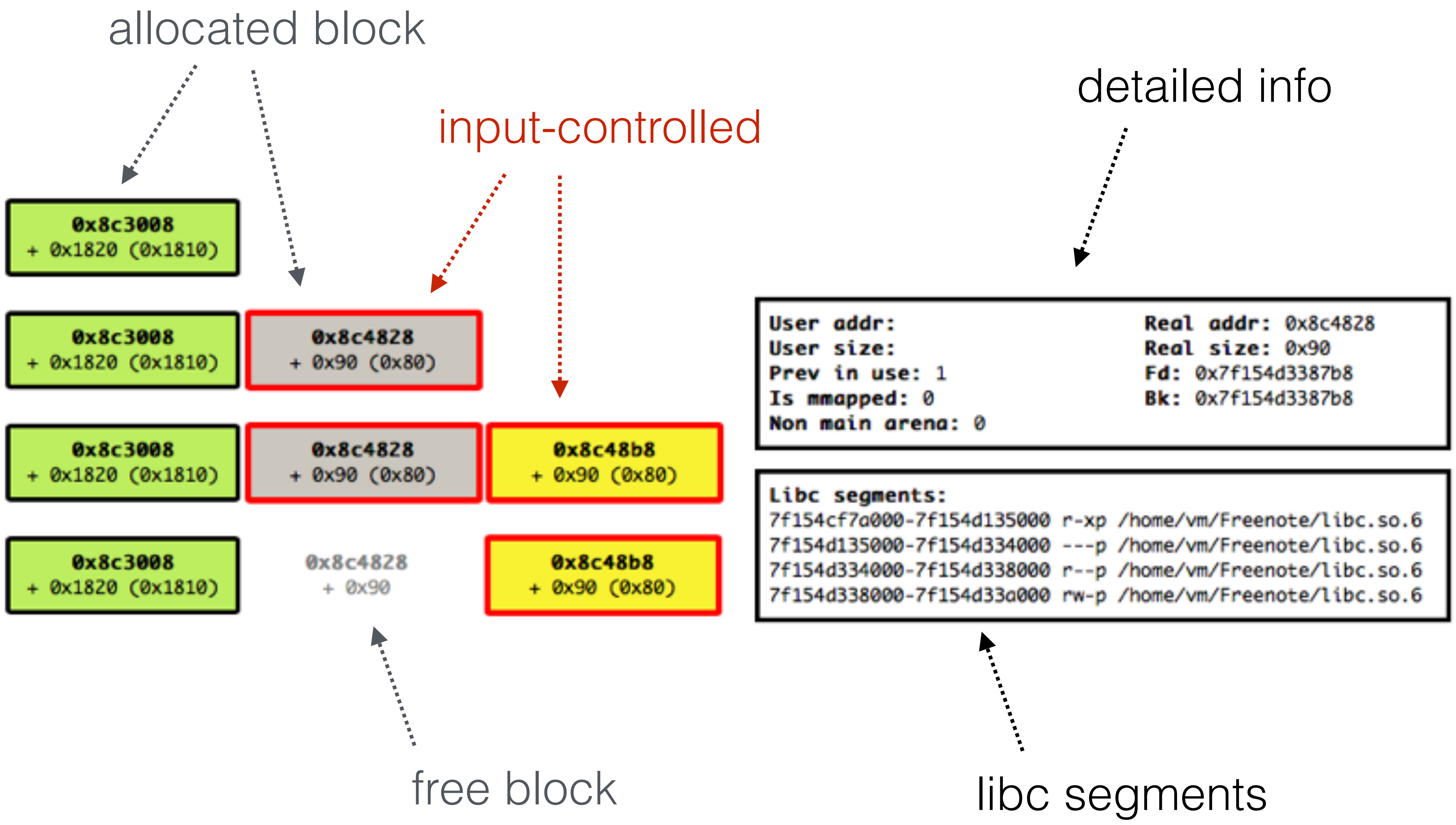




```
User addr:                    Real addr: 0x8c4828
User size:                    Real size: 0x90
Prev in use: 1                Fd: 0x7f154d3387b8
Is mmapped: 0                 Bk: 0x7f154d3387b8
Non main arena: 0
```

```
Libc segments:
7f154cf7a000-7f154d135000 r-xp /home/vm/Freenote/libc.so.6
7f154d135000-7f154d334000 ---p /home/vm/Freenote/libc.so.6
7f154d334000-7f154d338000 r--p /home/vm/Freenote/libc.so.6
7f154d338000-7f154d33a000 rw-p /home/vm/Freenote/libc.so.6
```

Graphical representation



```
User addr:                  Real addr: 0x8c4828
User size:                  Real size: 0x90
Prev in use: 1              Fd: 0x7f154d3387b8
Is mmapped: 0               Bk: 0x7f154d3387b8
Non main arena: 0
```

```
Libc segments:
7f154cf7a000-7f154d135000 r-xp /home/vm/Freenote/libc.so.6
7f154d135000-7f154d334000 ---p /home/vm/Freenote/libc.so.6
7f154d334000-7f154d338000 r--p /home/vm/Freenote/libc.so.6
7f154d338000-7f154d33a000 rw-p /home/vm/Freenote/libc.so.6
```

Experimental Evaluation

A Tests performed on CTFs (Capture The Flag), simple vulnerable programs used in IT Sec competitions.

PRO:

- Improved information gathering: visualize heap state changes and how program inputs affect them

- Direct feedback and debug of the operations performed during the exploitation

CONS:

- Considerable overhead due to binary instrumentation and symbolic execution (from 5x to 10x)

Future developments

- Developing an ad hoc framework for binary instrumentation and symbolic execution to reduce the overhead
- Extending the support to other architectures and allocators (e.g. ARM and jemalloc to test Android applications)

Contribute on GitHub

https://github.com/mauronz/HAIT

Contact me

andrea.marcelli@polito.it
jimmy-sonny.github.io/