KU LEUVEN

DISTRINET RESEARCH GROUP

HeapSentry

Kernel-assisted Protection against Heap Overflows

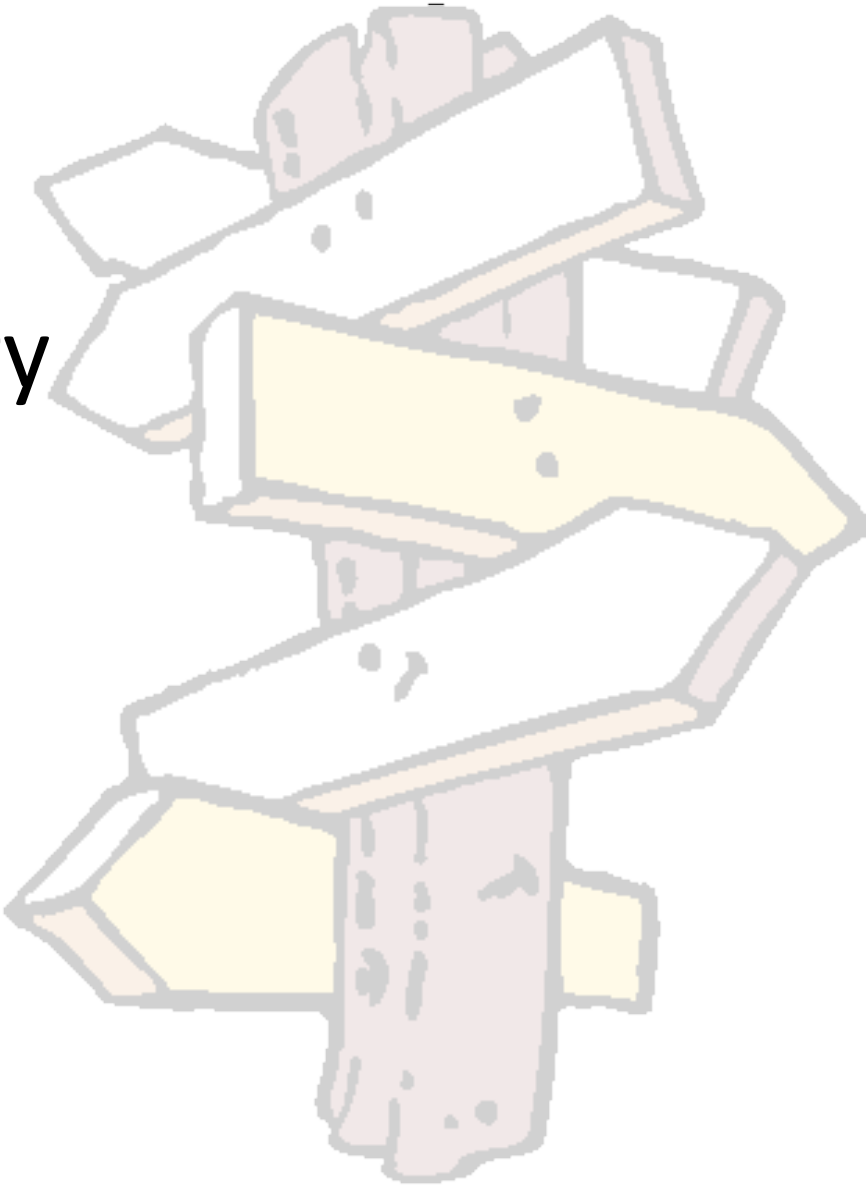
Nick Nikiforakis, Frank Piessens, Wouter Joosen

Sentry?

- *Sentry*: A soldier stationed to keep guard or to control access to a place

Roadmap

- Motivation
- Design of HeapSentry
- Evaluation
- Conclusion



Problem

- Overflows are still with us
  - 418 reported overflows for 2012
    - 135 were heap-based (32.3%)
  - 3rd in SANS Top 25 Most Dangerous Programming Errors
  - Most high-profile attacks usually involve one

Modern OSes

- Orthogonal runtime & compile-time countermeasures:
  - ASLR
  - W^X
  - Protection of stack frames
    - StackGuard, ProPolice

An old friend…

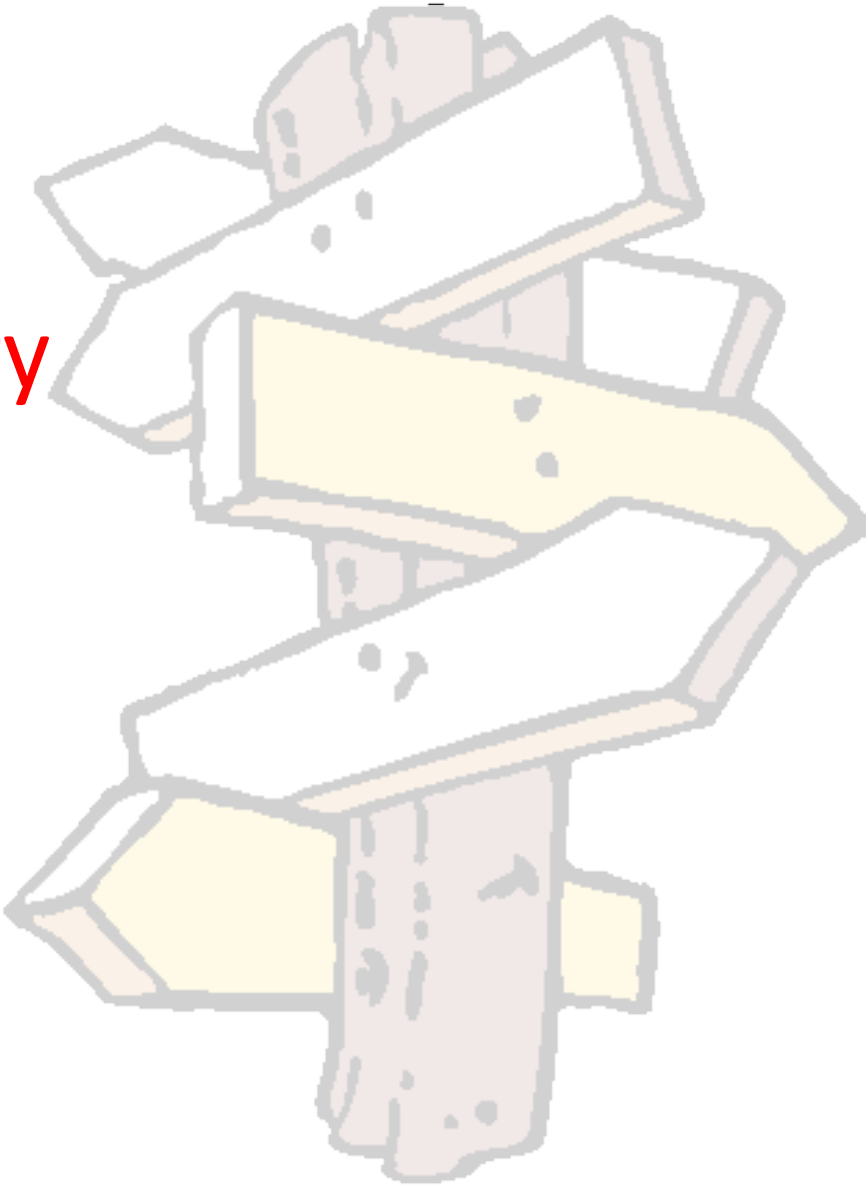
- The “Stack” part of a process used to be the most attacked one
  - Classic stack-smashing
  - Overwriting local args & function parameters
- Now its probably the least one
  - Canary in the way
  - Copied parameters and local args above the buffers

A “new” friend

- What about the heap?
  - Segment where all dynamic memory lives
- Who protects it?
  - No one….
    - most of the time
- An attacker can overflow freely from one heap object to the next
  - If he is detected, he will be detected at the next free … on that block

Roadmap

- Motivation
- Design of HeapSentry
- Evaluation
- Conclusion



HeapSentry characteristics

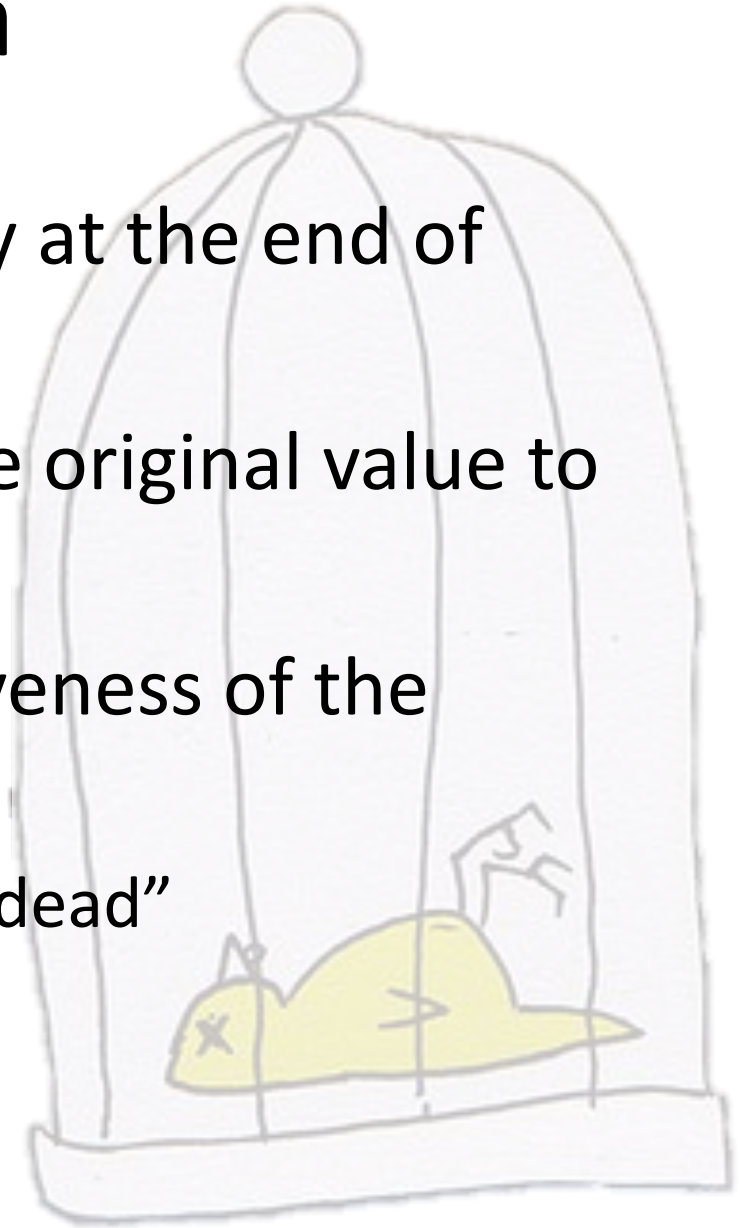
- Cooperative approach between a memory allocator and the OS' kernel to detect heap overflows
- Independent of underlying OS and memory allocator

Attacker model and requirements

- Independent of attacked object
  - Meta-data, function pointers, non-control data…
- Independent of attacking methodology
  - Injection of code, ret2libc, ROP-programming
- No need for source code

Core Idea

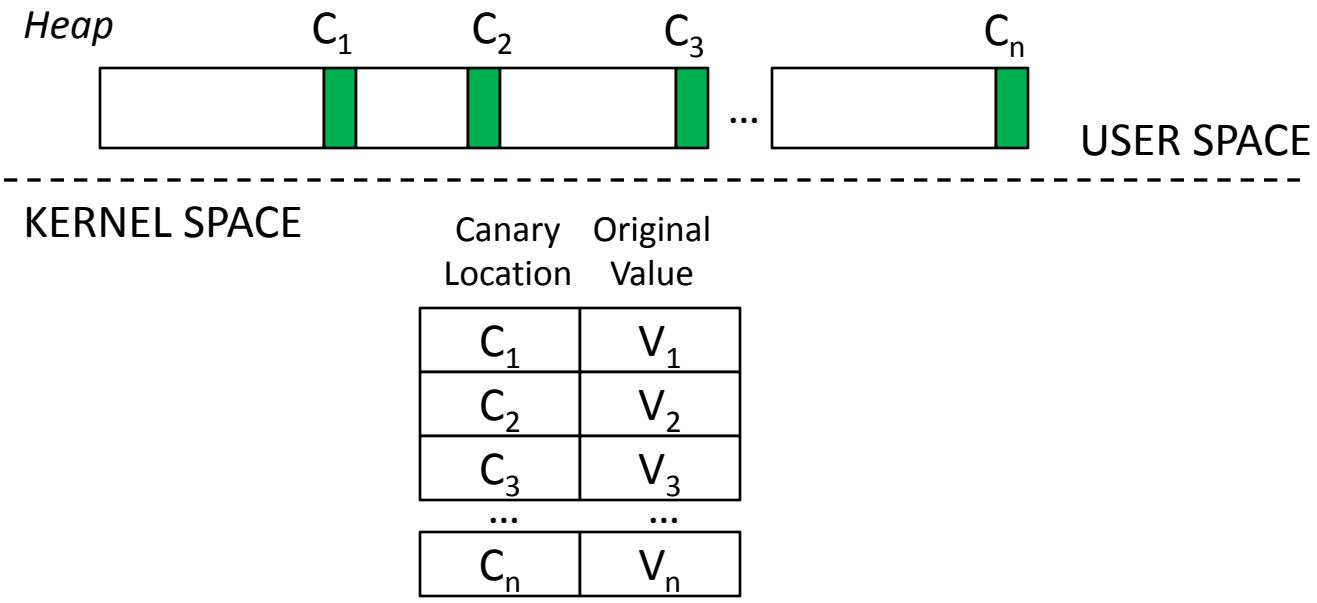
- Add a unique random canary at the end of each allocated block
- Register this location and the original value to the kernel
- Have the kernel check the liveness of the canaries
  - Kill the process if a canary is “dead”



Visually…

Heap

Visually…


Heap
C_1
C_2
C_3
C_n
USER SPACE
KERNEL SPACE
Canary Location
Original Value
C_1 V_1
C_2 V_2
C_3 V_3
C_n V_n

HeapSentry-U

- Intercept all calls to dynamic memory functions
  - malloc, calloc, realloc, free
- Generate random canaries and communicate to kernel

HeapSentry-U: malloc

- malloc(42)
  - ptr = real_malloc(42 + sizeof(int));
  - *(ptr + 42) = new random canary
  - syscall(ptr + 42) //HeapSentry-K
  - return to caller

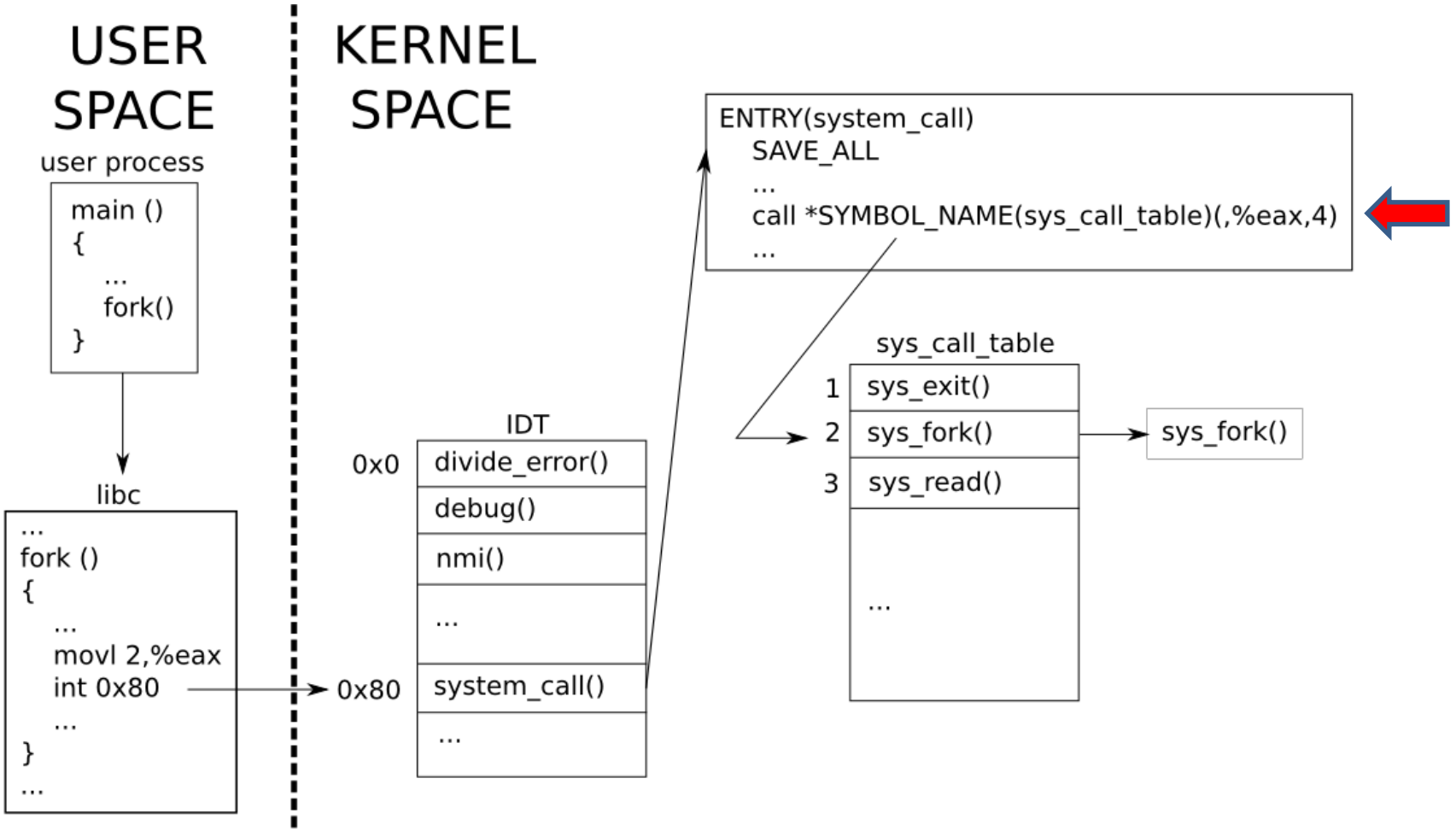
HeapSentry-U: free

- free(ptr)
  - canary_loc = ptr + sizeof(block) – sizeof(canary);
  - syscall(canary_loc);
  - real_free(ptr);
  - return to caller

HeapSentry-K

- Kernel module
  - Hijack the execution flow just before the dispatch of each individual system call
    - Exchange messages with HeapSentry-U
    - Check the liveness of canaries before the execution of a program-requested system call

Where?


USER SPACE
KERNEL SPACE
user process
main ()
{
...
fork()
}
libc
...
fork ()
{
...
movl 2,%eax
int 0x80
...
}
...
0x80
IDT
0x0
divide_error()
debug()
nmi()
...
system_call()
...
ENTRY(system_call)
SAVE_ALL
...
call *SYMBOL_NAME(sys_call_table)(,%eax,4)
...
sys_call_table
1
sys_exit()
2
sys_fork()
3
sys_read()
...
sys_fork()

Need for speed

- In its basic design, HeapSentry will execute one system call every time a dynamic memory function is called
  - Too much trapping to the kernel
  - Too many canaries to check at each system call

- Optimizations:
  1. System call categorization
  2. Grouping of operations

System call categorization

- The goal: Check less canaries at each system call invocation
- Are all system calls dangerous?
  - What is the chance that a system call is requested by an attacker
- Classes:
  1. High Risk – scan all
  2. Medium Risk – scan some
  3. No Risk – don't scan

Excerpt

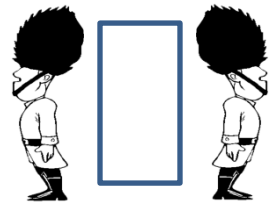
Category	System call
High Risk	fork execve chmod open
Medium Risk	read write mount mmap
No Risk	brk getpid chdir close

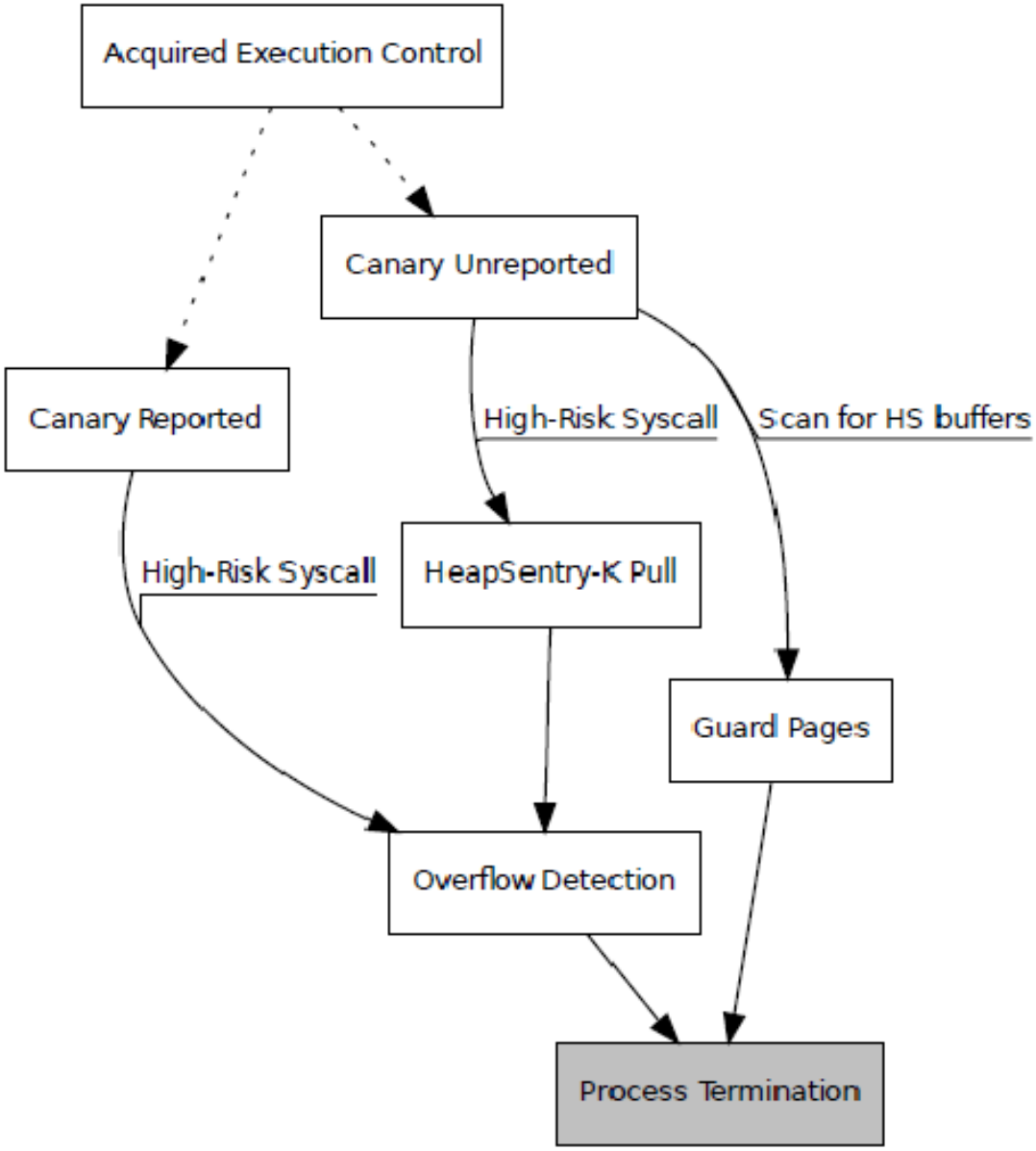
Grouping operations

- The goal: trap less times to the kernel
- Report to the kernel in batches instead of every time
  - e.g. here are 50 canaries and their values
  - e.g. check these 50 canaries so that I can free their blocks
- Result: 50 times less system calls over the basic HeapSentry design

Safety issues?

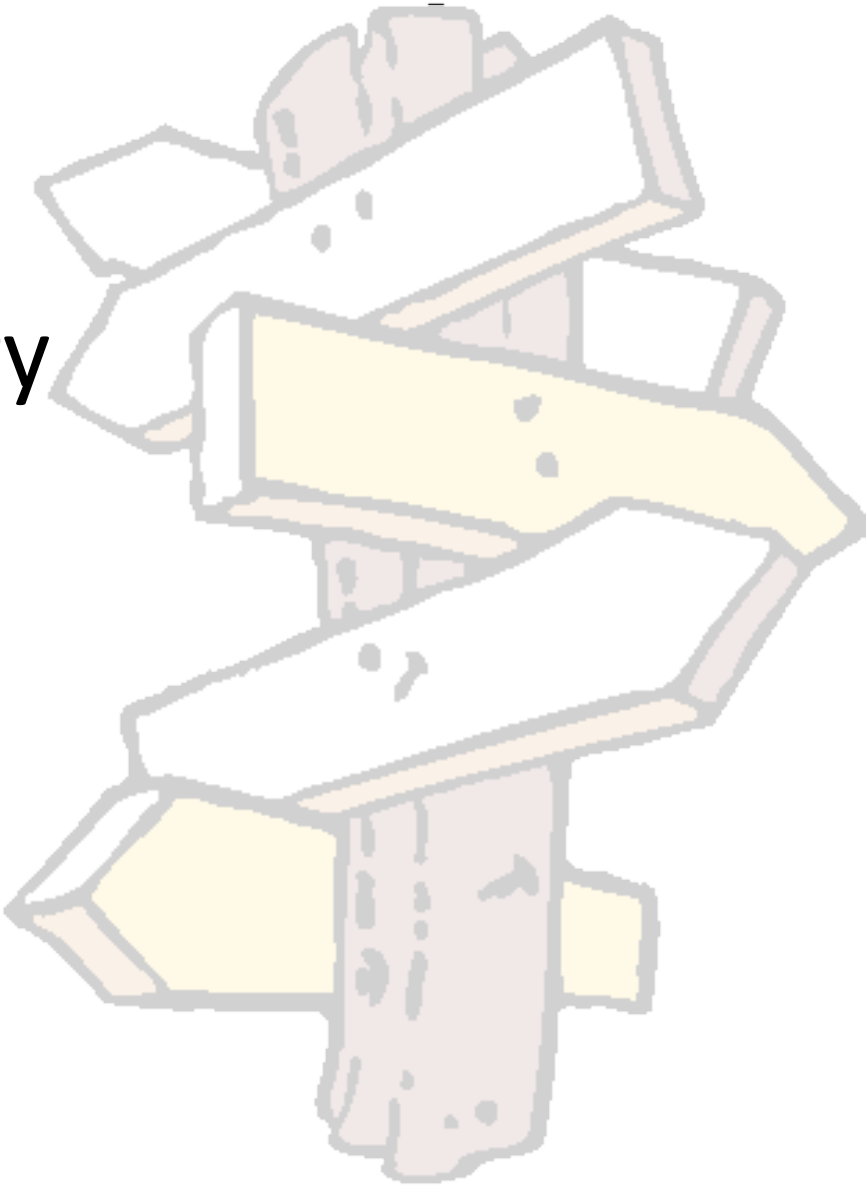
- Consider a memory-intensive program with an average of 100,000 active heap objects
- At any point in time:
  - a minimum of 99.95% of canaries are stored in the kernel
  - a maximum of 0.05% of canaries are still unreported
    - Attacker must locate buffers and remove before the process goes to the kernel
      - typical in multithreaded programs
    - Guard pages around the buffers




Acquired Execution Control
Canary Unreported
Canary Reported
High-Risk Syscall
Scan for HS buffers
High-Risk Syscall
HeapSentry-K Pull
Guard Pages
Overflow Detection
Process Termination

Roadmap

- Motivation
- Design of HeapSentry
- Evaluation
- Conclusion

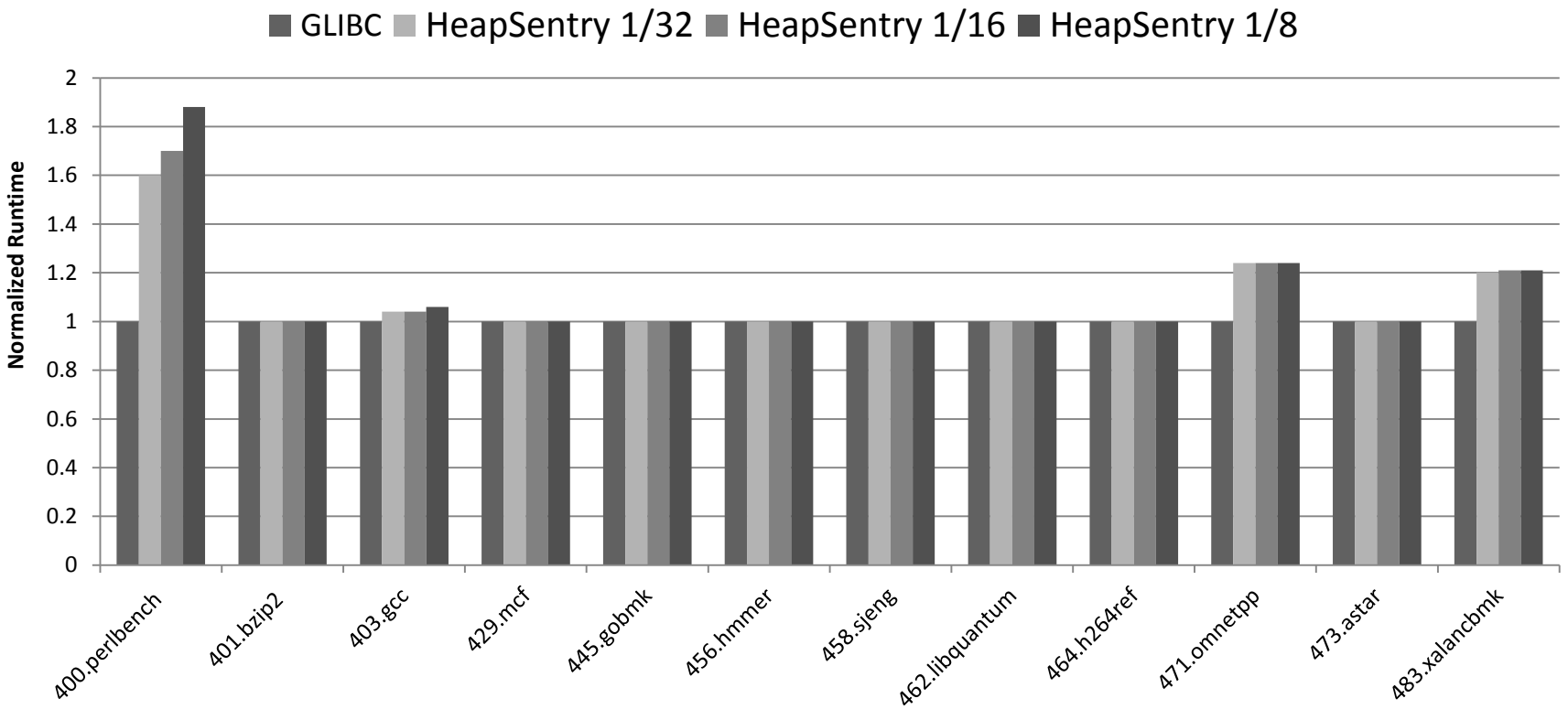


Security Evaluation

- Analyze 100 shellcode samples from shell-code.org
  - Removed 5 with non-critical payload
  - launching shells, killing IDSs, reading & editing system files
- All 95 where using at least one high-risk system call
  - HeapSentry would stop all of them in case of an attack

Performance

- SPEC CPU2006



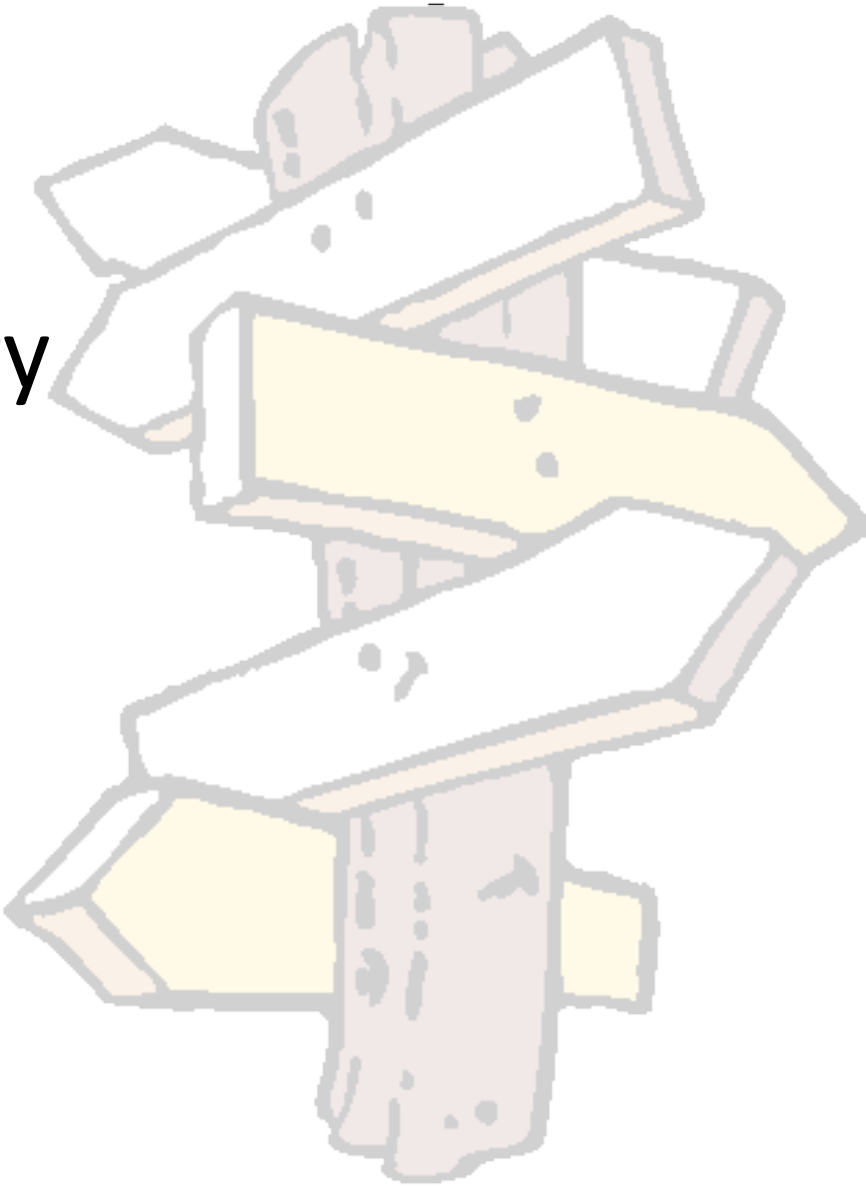
AVG: 11.6% added overhead

Limitations

- As with all canary-based approaches, if the canary is not overwritten, the heap overflow will not be detected
  - Overflow within the same allocated object
  - Direct overwrite

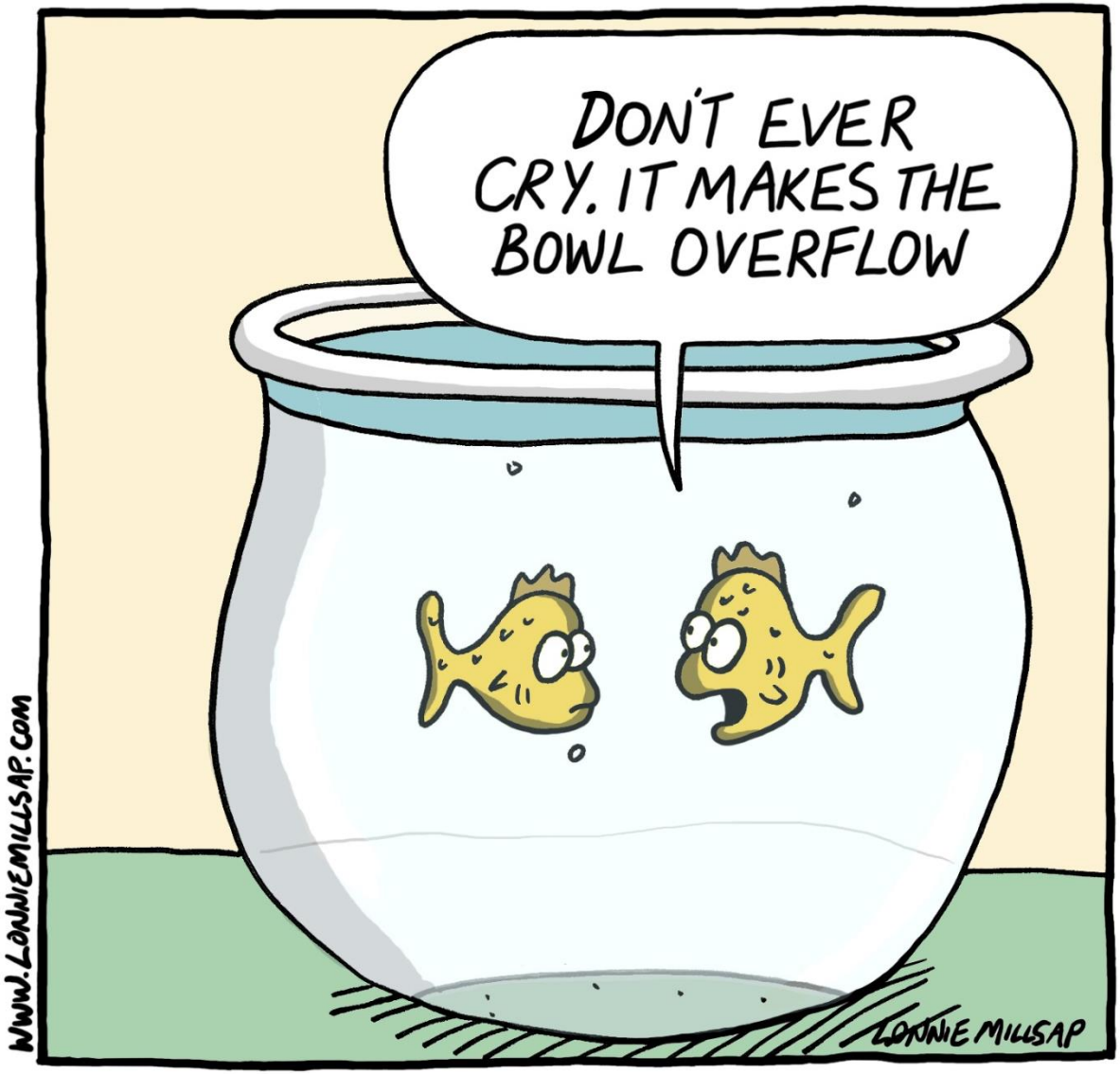
Roadmap

- Motivation
- Design of HeapSentry
- Evaluation
- Conclusion



Conclusion

- Buffer overflows are still an important problem
- The heap has received less attention than other parts of a process' address space
- HeapSentry, a cooperative approach between the memory allocation library and the kernel based on canaries



nick.nikiforakis@cs.kuleuven.be