

Preventing Use-After-Free Attacks with Fast Forward Allocation

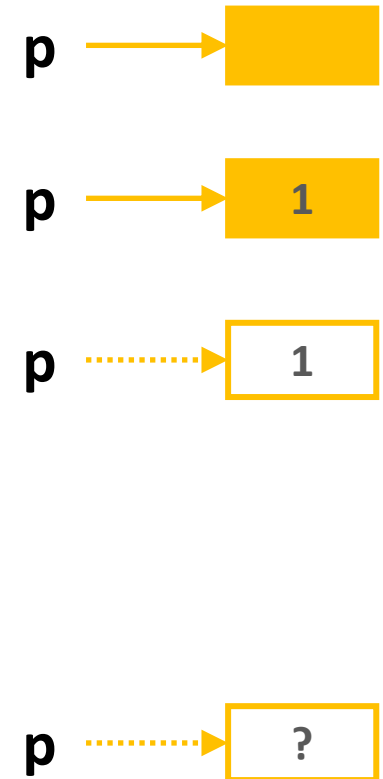
Brian Wickman,¹ Hong Hu,² Insu Yun,³ Daehee Jang,³ JungWon Lim,³
Sanidhya Kashyap,⁴ Taesoo Kim³

¹GTRI ²Penn State ³Georgia Tech ⁴EPFL

Use After Free

- A problem in memory unsafe languages like C
- Occurs when a program accesses memory it has previously marked as unused (free)
- If attackers can control this freed memory, normal program execution can be subverted

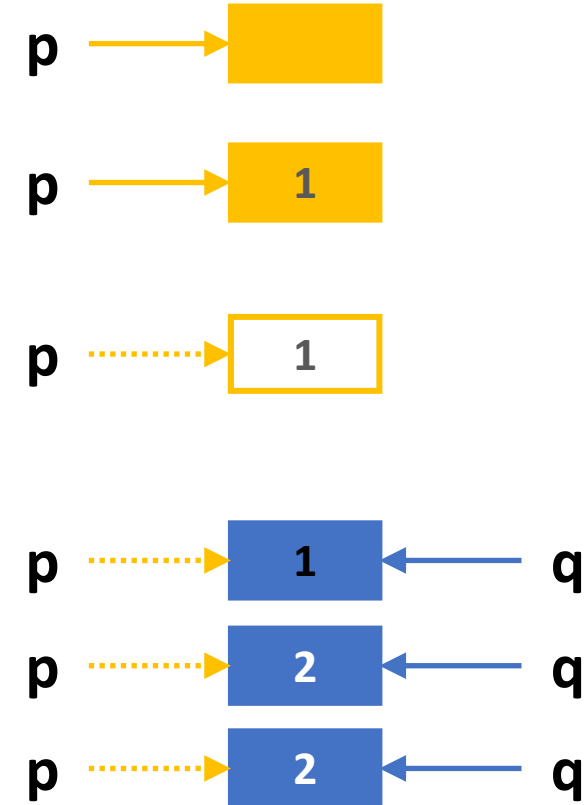
```
p = malloc(1);  
*p = 1;  
if (*p == 1) {  
    ...  
    free(p);  
}  
...  
...  
...  
if (*p == 2)  
...  
...
```



Attacking UAF

- Allocation
- Use
- Free
- Re-assignment
 - Most allocators reuse p's slot for q
- Use
- Use-after-free

```
p = malloc(8);
*p = 1;
if (*p == 1) {
    ...
    free(p);
}
...
q = malloc(8);
*q = ReadNet();
if (*p == 2)
    ...
```



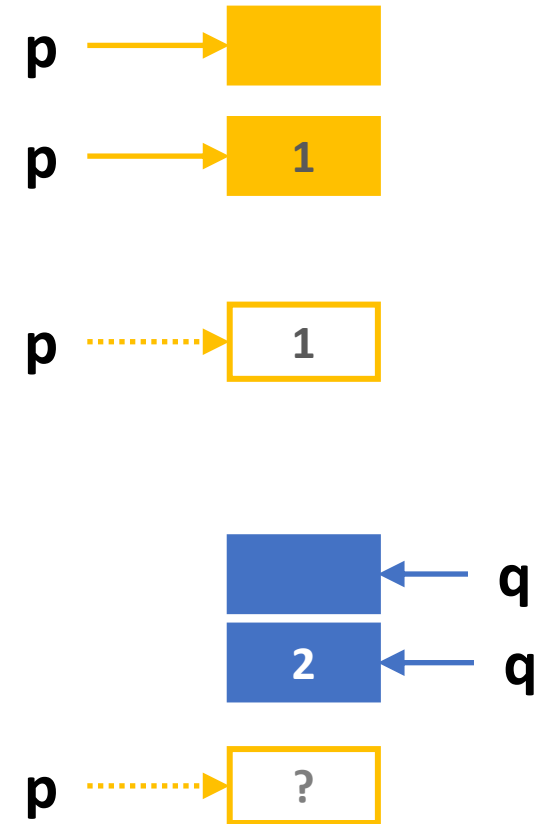
Still a Problem Despite Our Best Efforts

- Pointer invalidation – find and disable dangling pointers at runtime
 - Reference counting
 - Garbage collection inspired searching
- UAF detection – compiler injected runtime checks
- Allocation randomization – don't reuse addresses ... probably
 - Hopefully the attacker's target doesn't get reassigned too quickly
- Restricted reuse
- Page-per-allocation – immediately unmap VAs upon free
- None are widely adopted

(Re)Introducing One Time Allocation

- What if we just never* re-use VAs?
 - Doesn't remove the UAF bug
 - Does makes it unexploitable
- The naïve and simplest approach
 - No expensive tracking
 - No custom compilers
- Previously thought impractical
 - We needed 64-bit address space
 - What about CPU overhead?
 - Won't memory be wasted?

```
p = malloc(8);
*p = 1;
if (*p == 1) {
    ...
    free(p);
}
...
q = malloc(8);
*q = ReadNet();
assert(p != q);
if (*p == 2)
    ...
```



Take 1 - Forward Continuous malloc

- Simple bump pointer allocator
- Address space fragmentation
- Small long-lived allocations prevent releasing pages
- Exhausts VMAs
- This is what early objections to OTA warned about
- Key lesson learned – Use batch page release to decrease VMA and CPU usage albeit for an increase in memory consumption

Benchmark	glibc	FCmalloc
perlbench	4,401	58,737
bzip2	23	35
gcc	2,753	6,525
mcf	20	31
milc	46	65
namd	128	57
gobmk	25	61
dealII	4,760	2,322
soplex	152	99
povray	51	109
hmmer	35	197
sjeng	20	32
libquantum	29	38
h264ref	228	89
lbm	23	34
omnetpp	1,164	15,933
astar	1,762	6,726
sphinx3	168	31,409
xalancbmk	2,705	68,606

Take 2 - Forward Binning malloc

- Fit allocations into fixed sized buckets
- Put allocations of the same size onto the same pages
- Reduces VMA pressure. Long lived allocations more likely to live together
- Larger allocations round up to page boundary
- Potentially significant memory waste

Take 3 - FFmalloc – The best of both

From FCmalloc

- Allocations > 2048 bytes
- Processor required alignment only – minimize allocation waste
- Tunable “consecutive pages before release” parameter to control CPU vs memory consumption

From FBmalloc

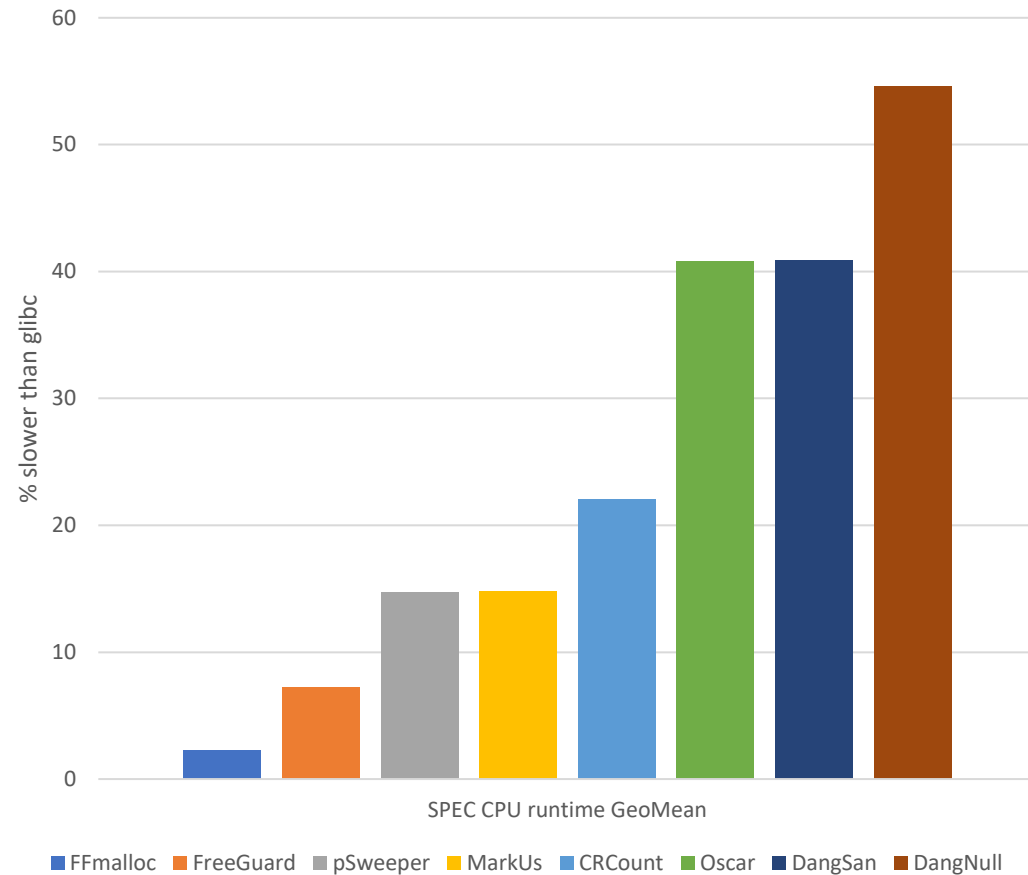
- Smaller allocations grouped into fixed sized buckets
- Long lived small allocations don't block page release
- Small allocations never cross page boundaries

It's Effective

Program	ID	Bug Type	Link	Original Attack	With the Protection of FFmalloc
babyheap	CTF challenges	UAF → DF	[10]	✓ Arbitrary code execution	✗ Exception due to failed info-leak
childheap		UAF → DF	[10]	✓ Arbitrary code execution	✗ DF detected
heapbabe		UAF → DF	[1]	✓ Arbitrary code execution	✗ DF detected
ghostparty		UAF	[9]	✓ Arbitrary code execution	✗ Exception due to failed info-leak
uaf		UAF	[8]	✓ Arbitrary code execution	✗ Exploit prevented due to new realloc
PHP 7.0.7	CVE-2016-5773	UAF → DF	[7]	✓ Arbitrary code execution	✗ Exploit prevented & DF detected
PHP 5.5.14	CVE-2015-2787	UAF	[6]	✓ Arbitrary code execution	✗ Assertion failure (uncontrollable)
PHP 5.4.44	CVE-2015-6835	UAF	[5]	✓ Arbitrary memory disclosure	✗ Exploit prevented & run well
mruby 191ee25	Issue 3515	UAF	[23]	✓ Arbitrary memory write	✗ Exploit prevented & run well
libmimedir 0.5.1	CVE-2015-3205	AF → UAF	[15]	✓ Arbitrary code execution	✗ Exploit prevented & run well
python 2.7.10	Issue 24613	UAF	[28]	✓ Restricted memory disclosure	✗ Exploit prevented & run well

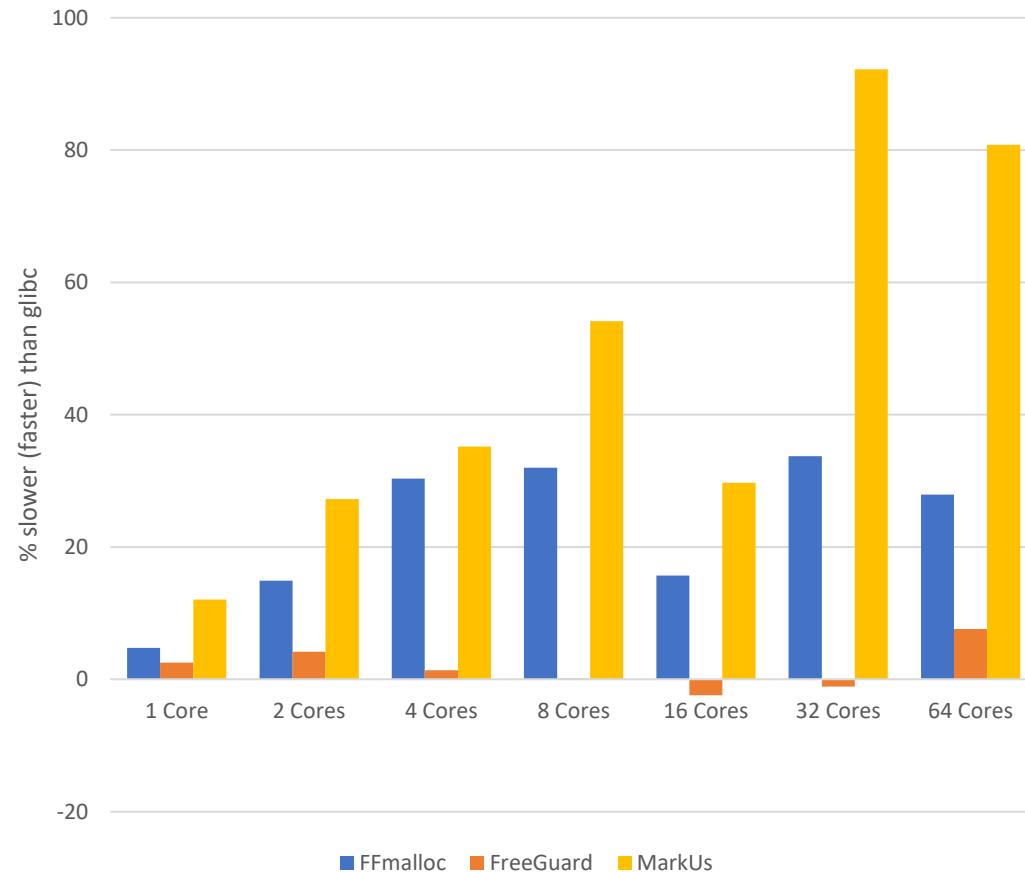
UAF: Use-After-Free, **DF**: Double Free, **AF**: Arbitrary Free

Minimal Single Thread CPU Overhead



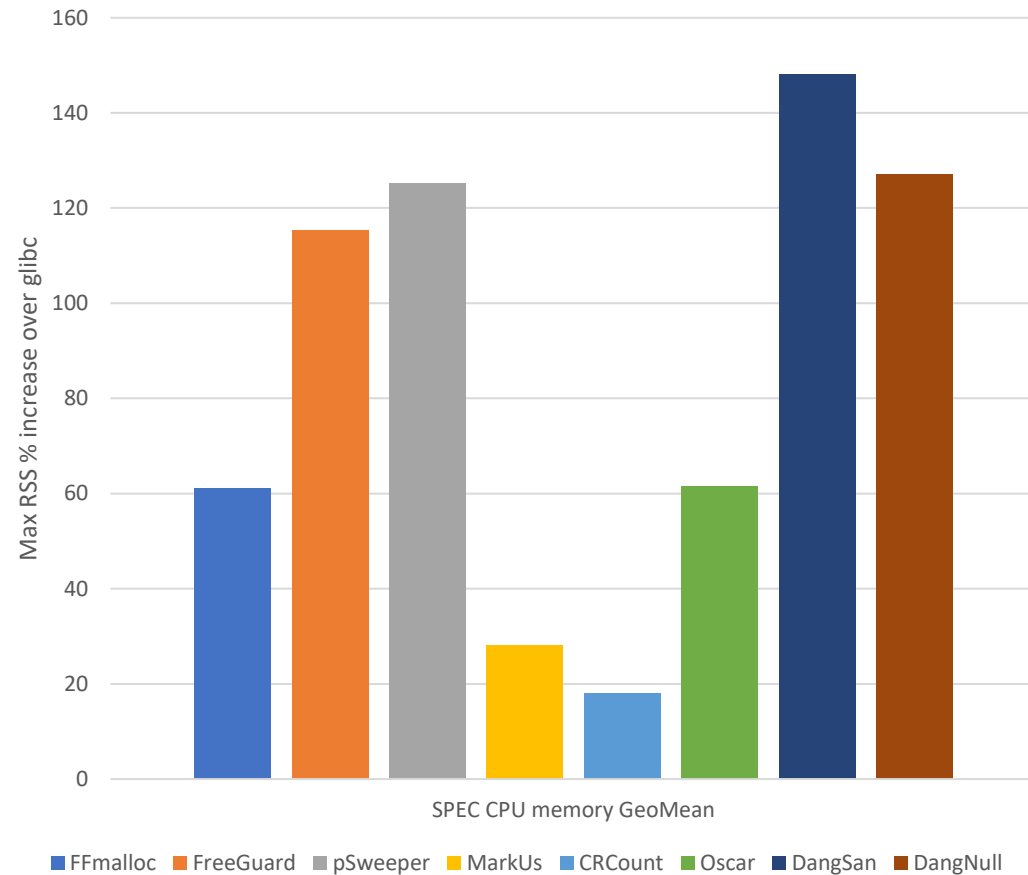
- Contrary to intuition, one-time-allocation can perform well.
- FFmalloc out-performed seven previous UAF defense proposals on SPEC benchmarks
- Added only 2.3% overhead to SPEC benchmarks (geometric mean)

Low Multi-thread CPU Overhead



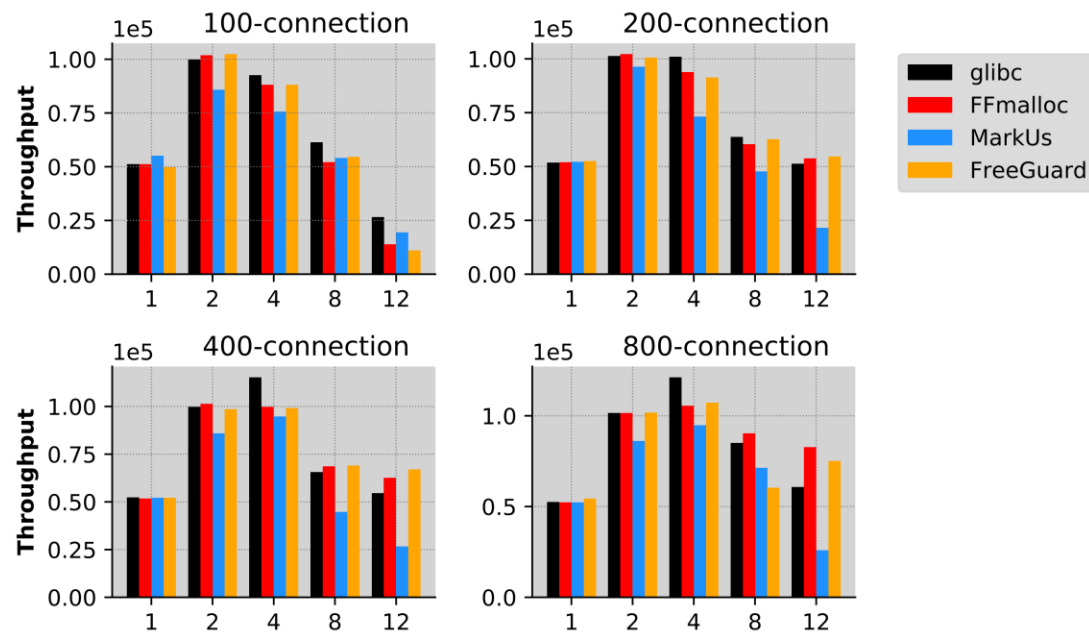
- On PARSEC, FFmalloc added 22% overhead versus 43% for MarkUs or 1.7% for FreeGuard
- mmap-sem lock in the Linux kernel constrains FFmalloc
 - m(un)map calls on parallel threads get serialized

Moderate Memory Overhead



- FFmalloc is neither the best nor worst in terms of imposed memory overhead
- FFmalloc could release pages more aggressively at the cost of additional CPU usage.

Nginx Load Testing




- Comparable throughput (requests/second serviced) to glibc on an Nginx webserver
- Memory utilization was high at 5.24x glibc but comparable to FreeGuard and much better than MarkUs

Contrasts with Related Work

- Probabilistic reuse may be of limited value
 - Multiple chances if bug is network visible
 - Hard to reason about
- Pointer tracking generally too expensive
- Does not require recompiling
- FFmalloc has a hard guarantee that is easy to reason about

Summary

- UAF bugs are still significant.
 - Vulnerable code bases include operating systems, browsers, and even the runtimes of many memory safe languages
- One-time-allocation is effective and simple to implement
- Concerns about OTA CPU and memory inefficiency can be addressed through smart design



Contact Information



FFmalloc published at
<https://github.com/bwickman97/ffmalloc>



Questions?
brian.wickman@gtri.gatech.edu (please
include “ffmalloc” in your subject line)