Crowdsourced bug detection in production: GWP-ASan and beyond

Matt Morehouse, Mitch Phillips, Kostya Serebryany

C++ Russia / November 2020

Agenda

- C++ memory safety landscape
- GWP-ASan, a sampling-based bug detector for production
  - Algorithm
  - Deployment
- Can GWP-ASan become a security mitigation?
- Applying the same approach to other bug classes?
- A few words about Arm Memory Tagging Extension (Arm MTE)

Memory safety bugs, sanitizers, fuzzing, hardening

- Use-after-free, buffer-overflow, etc: > 50% of CVEs across the industry [1] [2]
- Static analysis: useful, but misses many cases
- Dynamic analysis (e.g. ASan, HWASan, Valgrind):
  - Finds everything that happens in a test, but tests cover too little
  - Production deployment near-impossible due to overheads (some still do it, “prod canaries”)
- Fuzzing (libFuzzer, AFL, Syzkaller, etc): improves test coverage
  - Finds 10x more bugs than testing, but still not everything
- Hardening (e.g. Control Flow Integrity, hardened malloc, etc)
  - Blocks certain exploitation techniques (e.g. ROP); does not address the root cause
- (near future) Hardware extensions like Arm MTE: detect bugs in production
  - Huge step forward, but will not be available everywhere for many years

What is GWP-ASan?

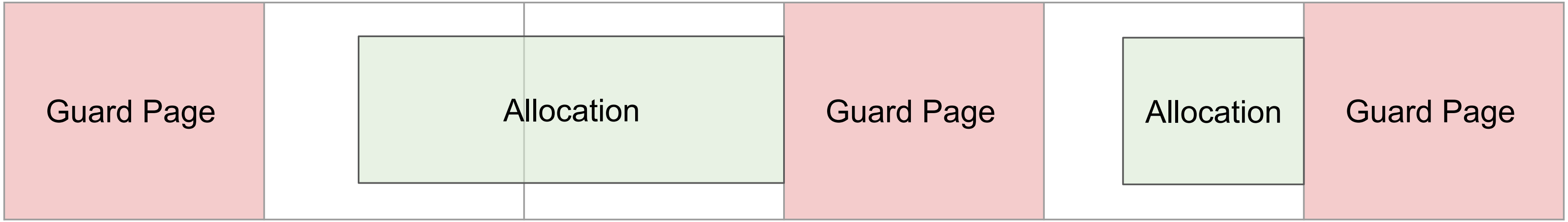
- "**G**WP-ASan **W**ill **P**rovide **A**llocation **Sa**nitization"

- Also: GWP: Google-Wide Profiling + ASan: AddressSanitizer
  - GWP-ASan is neither GWP nor ASan, but the name reflects well what it is.

- Probabilistic memory safety error detector (heap only)
  - Detects heap-buffer-overflow and heap-use-after-free.

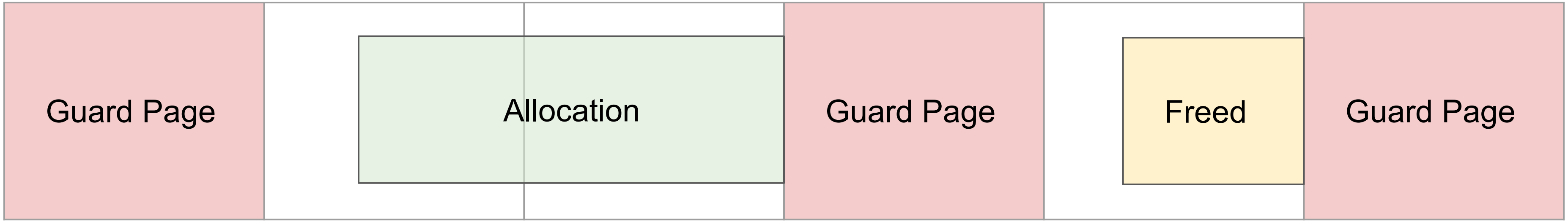
Background: Electric Fence

- Detects heap-buffer-overflows using **guard pages.**



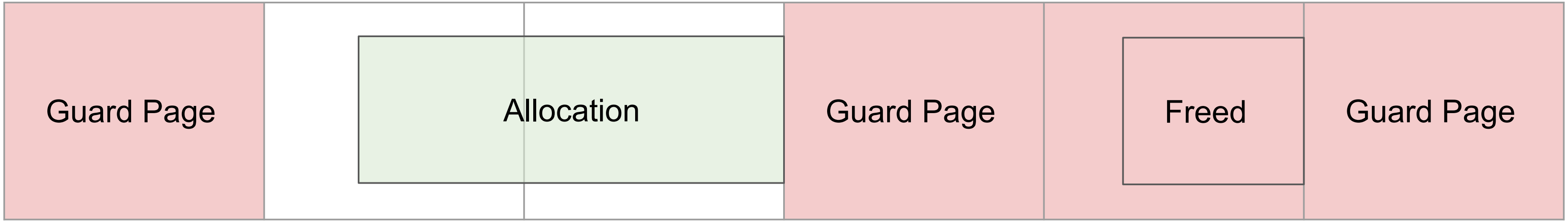
Background: Electric Fence

- Detects heap-buffer-overflows using **guard pages.**
- Detects use-after-frees by *mprotect*-ing freed memory**.**



Background: Electric Fence

- Detects heap-buffer-overflows using **guard pages.**
- Detects use-after-frees by *mprotect*-ing freed memory**.**



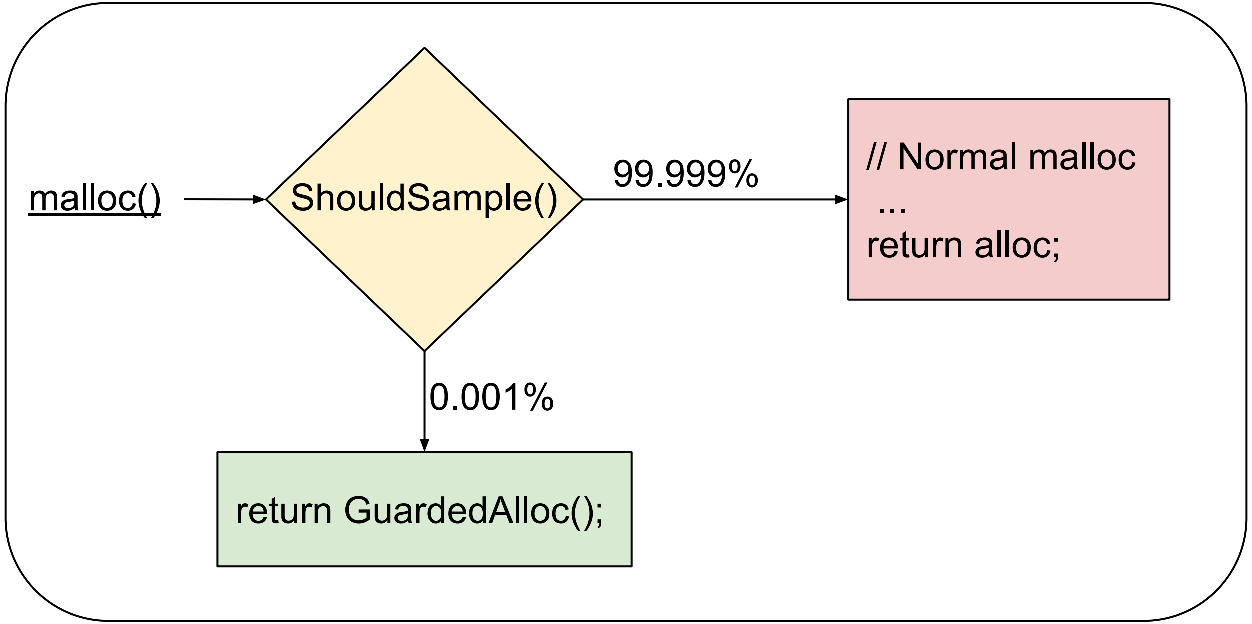
Background: Electric Fence (since ~ 1987)

- + Detects bugs!
- + No compiler instrumentation.
    - Can be enabled at link-time (-lefence) or runtime (LD_PRELOAD=/usr/lib/libefence.so.0).

- - Really expensive.
    - **Heap fragmentation (~100x)**
        - Each allocation needs a full 4KB page for buffer overflow detection.
    - **Slow (~100x)**
        - Most mallocs and frees require a system call to *mmap* or *mprotect*.

GWP-ASan = Electric Fence + Sampling

- Randomly guard a tiny fraction of allocations (e.g. 1/100,000).
  - Make overhead as low as we want.



GWP-ASan at Google

Deployment Status

- Chrome: **on-by-default**
  - for Windows and macOS only
  - Example bug
- Google server-side applications: **on-by-default**
- Android R: **on-by-default** for system processes, opt-in for apps
  - Developing an Android App with native code? Please try it now!
- Linux Kernel: coming soon (kfence)

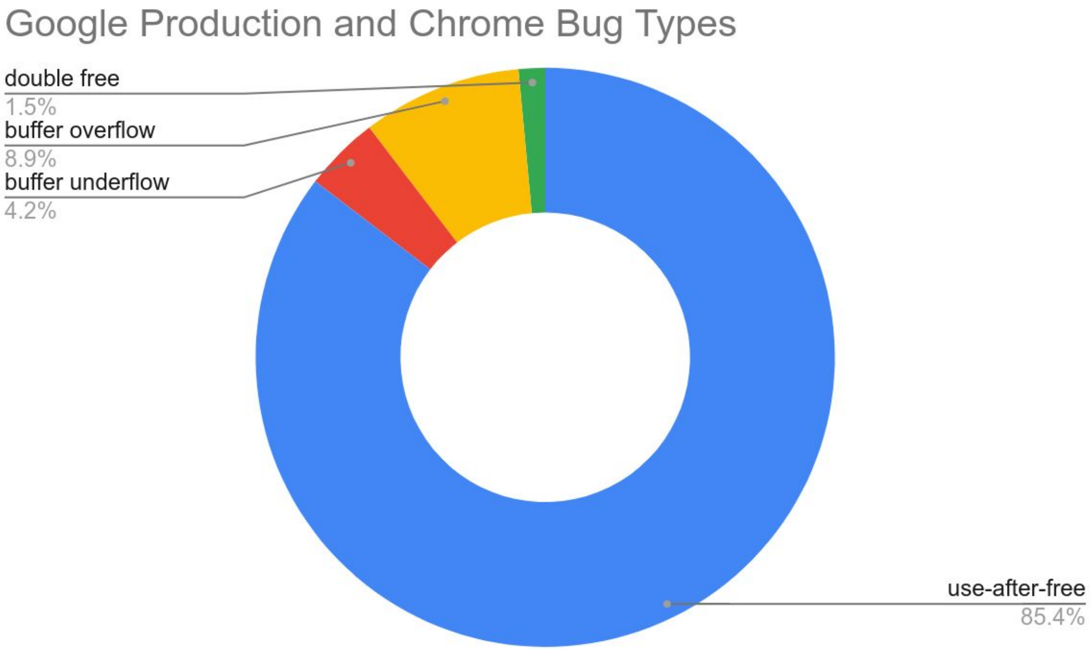
Results

- Chrome
    - [140+ bugs]

      over past ~ 1.5 years.
- Google Production
    - **2000+ bugs**

      over past ~ 1.5 years.
- Android R
    - 80+ bugs

      Just started



Using GWP-ASan

Available in LLVM

- GWP-ASan lives in LLVM / compiler-rt.
- Comes with Scudo Hardened Allocator (*-fsanitize=scudo*)*
    - * x86/x86_64 only
- Simple integration with any other memory allocator.

Scudo Example

```
$ cat buggy_code.cc
#include <iostream>
#include <string>
#include <string_view>

int main() {
  std::string s = "Helloooooooooooooo ";
  std::string_view sv = s + "World\n";
  std::cout << sv;
}
$ clang++ -g -std=c++17  -fsanitize=scudo buggy_code.cc && ./a.out
Helloooooooooooooo World
$ for((i=0; i<1000; i++)); do GWP_ASAN_OPTIONS=SampleRate=500 ./a.out >/dev/null | symbolize.sh; done
*** GWP-ASan detected a memory error ***
Use after free at 0x7fb4b941e000 (0 bytes into a 41-byte allocation at 0x7fb4b941e000) by thread 140162 here:
  ...
  #9 /usr/lib/gcc/x86_64-linux-gnu/8.0.1/../../../../include/c++/8.0.1/string_view:547
  #10 /tmp/buggy_code.cpp:8

0x7f76bb8bafd0 was deallocated by thread 103932 here:
  ...
  #7 /tmp/buggy_code.cpp:8
```

Integrating with a Memory Allocator

```
static gwp_asan::GuardedPoolAllocator GuardedAllocator;

void initMalloc() {
  ...
  gwp_asan::options::Options Opts = …  // Configure as desired.
  GuardedAllocator.init(Opts);
}

void *malloc(size_t Size) {
  ...
  if (PREDICT_FALSE(GuardedAllocator.shouldSample()))
    if (void *Ptr = GuardedAllocator.allocate(Size))
      return Ptr;
  ...
}

void free(void *Ptr) {
  ...
  if (PREDICT_FALSE(GuardedAllocator.pointerIsMine(Ptr)))
    return GuardedAllocator.deallocate(Ptr);
  ...
}
```

Also available via …

- TCMalloc:
  - https://github.com/google/tcmalloc/blob/master/docs/gwp-asan.md

- Chromium:
  - https://chromium.googlesource.com/chromium/src/+/master/docs/gwp_asan.md

Small print

- GWP-ASan itself is small and simple
    - You can use one of our implementations: TCMalloc, LLVM, or Chrome
    - Or you can implement your own, like Mozilla does

- The hardest part is the bug reporting pipeline, which will be project-specific
    - Collect, symbolize, aggregate the reports
    - Track them over time, confirm fixes
    - Ignore false positives (e.g. due to cosmic rays - no kidding)
    - But, don’t ignore one-off reports
    - Make sure user privacy is not compromised

GWP-ASan as part of the developer workflow

- GWP-ASan is the last resort, you better find bugs the other ways
    - A.k.a. “Shift Left”

- Learn from GWP-ASan reports:
    - Focus your fuzzing on components with GWP-ASan reports
    - Do the postmortems, figure out why the bug crept into production
    - Make sure you have regression tests for all fixes
    - Find common bug patterns and handle them statically
        - E.g. -Wdangling-gsl handles some of the cases with ~~std::use_after_free~~ std::string_view

Can GWP-ASan become a security mitigation?

- Not a hardening tool in the usual sense
  - 99999 attack attempts out of 100000 will succeed
  - Better if an exploit chains multiple memory safety vulnerabilities

- But does this change the economics for exploit developers?
  - Today: exploit development is expensive, the same exploit is used on many targets for a long period of time
  - With wider use of GWP-ASan: the cost remains the same, the number of successful attacks before detection drops. Unstable exploits become even less usable
- What can we do to make the detection
  - more likely?
  - less predictable?

Research: improve detection w/o increasing cost

- Help is welcome!

- Statistical tricks, e.g. guard the least frequent allocations?

- Machine learning? What allocations are more likely to be involved in a use-after-free or buffer overflow?

What about other bug classes?

- UBSan-like checks (e.g. integer overflows, etc)
  - Existing solutions rely on compiler instrumentation
  - Can't easily sample at run-time
  - Maybe use debug registers to stop at arbitrary instructions?
  - Maybe sample at compile time (cover all the code eventually, assuming frequent releases)
- Stack-buffer-overflow, use-after-return
  - Don't know how to detect w/o compiler instrumentation
- Use of uninitialized memory
  - Perhaps, easier/cheaper to just initialize everything (work in progress)
- Anything else?

- Data races: coming soon, stay tuned!

Memory Tagging: Arm MTE

Arm Memory Tagging Extension (MTE)

- Announced by Arm on 2018-09-17

- Doesn't exist in hardware yet
  - Will take several years to appear

- "Hardware-ASAN on steroids"
  - RAM overhead: 3%-5%
  - CPU overhead: (*hoping for)* low-single-digit %

ARM Memory Tagging Extension (MTE)

- 64-bit only
- Two types of tags
  - Every aligned 16 bytes of memory have a 4-bit tag stored separately
  - Every pointer has a 4-bit tag stored in the top byte
- LD/ST instructions check both tags, raise exception on mismatch
- New instructions to manipulate the tags

Allocation: tag the memory & the pointer

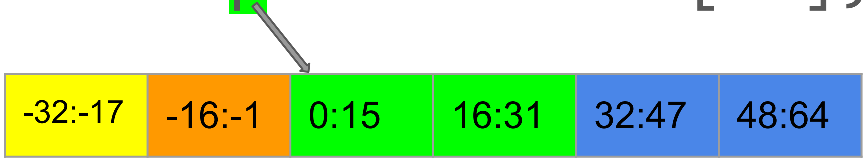
- Stack and heap

- Allocation:
  - Align allocations by 16
  - Choose a 4-bit tag (random is ok)
  - Tag the pointer
  - Tag the memory (optionally initialize it at no extra cost)

- Deallocation:
  - Re-tag the memory with a different tag

Heap-use-after-free

```
char *p = new char[20]; // 0xa007fffffff1240
```



Heap-use-after-free

```
char *p = new char[20]; // 0xa007fffffff1240
```

-32:-17	-16:-1	0:15	16:31	32:47	48:64

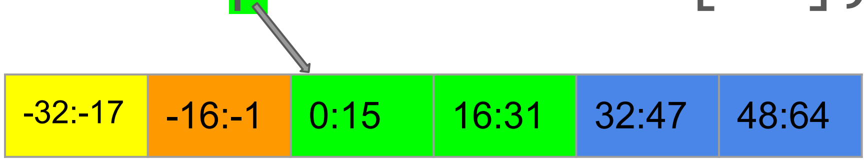
```
delete [] p;  // Memory is retagged ⇒
```

-32:-17	-16:-1	0:15	16:31	32:47	48:64

```
p[0] = …  // heap-use-after-free ≠
```

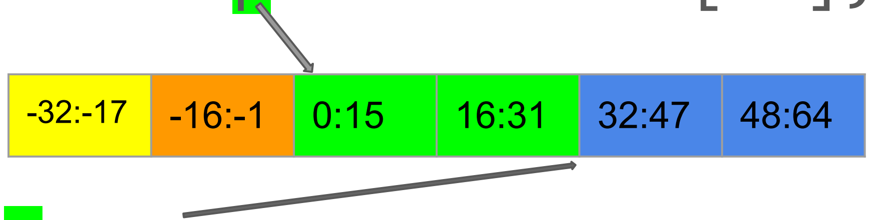
Heap-buffer-overflow

```
char *p = new char[20]; // 0xa007fffffff1240
```



Heap-buffer-overflow

```
char *p = new char[20]; // 0xa007fffffff1240
```




```
p[32] = … // heap-buffer-overflow
```

≠

Probabilities of bug detection

```
int *p = new char[20];

p[20]                 // undetected, same granule (*)

p[32], p[-1]          // 93%-100% (15/16 or 1)

p[100500]             // 93% (15/16)

delete [] p; p[0]     // 93% (15/16)
```

Buffer overflows within a 16-byte granule

- Typically, not security bugs if heap/stack is 16-byte aligned in production

- Still, logical bugs

- Only so-so solutions for testing:
  - Malloc may optionally align right (tricky on ARM, more tricky on x86_64)
  - Put magic value on malloc, check on free (detects only overwrites, with delay)
  - Tag the last granule with a different tag, handle in the signal handler (SLOW)

MTE Overhead

- RAM: 3% - 5% (measured)
- Code Size: 2%-4% (measured)
- CPU: 0% - 5% (*estimated*)
- Power: ?

MTE Usage Models

- Testing in lab
  - Better & cheaper than ASAN

- **Testing in production** aka crowdsourced bug detection
  - possibly with per-process or per-allocation sampling
  - actionable deduplicated bug reports

- Always-on **security mitigation**
  - with per-process knobs

Is probabilistic detection OK for security mitigation?

- Enough retries may allow an MTE bypass in some cases (e.g. UAF)

- BUT:
  - Software could block the restarts on first MTE report (i.e. no retries)
  - The vendors gets actionable bug report on first failed attempt
  - Extra security layers can be built on top of MTE (e.g. MarkUs-GC)

Legacy code

- MTE will work on legacy code w/o recompilation
  - Libc-only change
  - Will find and mitigate heap OOB & UAF (~90% of all bugs)

No more uses of uninitialized memory

- Tagging the memory during allocation also initializes it
  - MTE always-on => no more uninitialized memory
  - MTE only during testing => uninitialized memory remains

- Can initialize all memory today, at ~ the same cost as full MTE

Try GWP-ASan today, ask your CPU vendor for MTE

- GWP-ASan:
    - chromium.googlesource.com/chromium/src.git/+/master/docs/gwp_asan.md
    - github.com/google/tcmalloc/blob/master/docs/gwp-asan.md
    - developer.android.com/ndk/guides/gwp-asan
    - llvm.org/docs/GwpAsan.html
    - llvm.org/devmtg/2019-10/talk-abstracts.html#lit1

- Arm MTE:
    - 2019-08-02 Android blog post
    - 2019-08 Arm whitepaper
    - Security analysis by Microsoft.