Beyond Sanitizers: Guided fuzzing and security hardening

LLVM Developer's Meeting, Oct 29, 2015

Kostya Serebryany kcc@google.com
Peter Collingbourne pcc@google.com

Agenda

- Sanitizers: dynamic testing tools for C++
  - ASan, TSan, MSan, UBSan
- Fuzz testing
- Code hardening

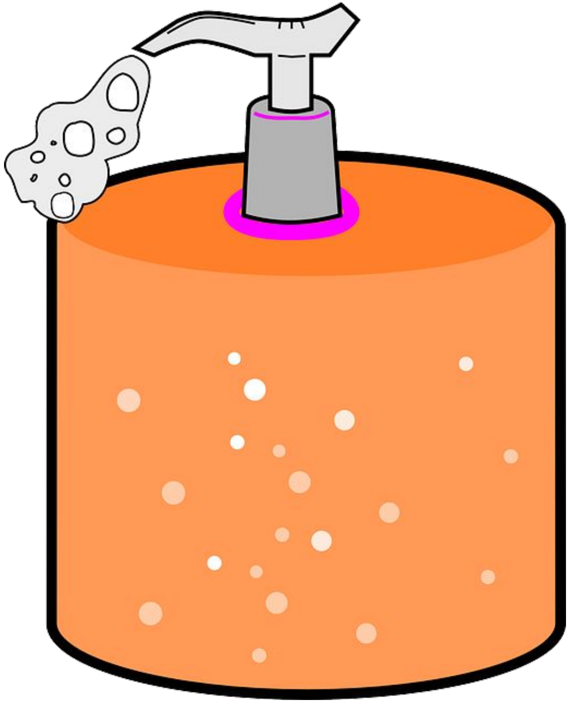
C++: shoot yourself in the ~~foot~~ feet

- Buffer overflow (heap, stack, global)
- Heap-use-after-free, stack-use-after-return
- Data race, deadlock
- Use of uninitialized memory
- Memory leak
- Integer overflow
- ...

Why do you care?

- Hard to reproduce and debug
- Sporadic crashes or data corruption
- Excessive resource consumption
- Wrong results
- ...

SECURITY



Sanitizers

Dynamic Testing Tools for C++
based on compile-time instrumentation

ASan
(AddressSanitizer)

Bugs related to addressing memory

ASan report example: global-buffer-overflow

```
int global_array[100] = {-1};
int main(int argc, char **argv) {
  return global_array[argc + 100];  // BOOM
}
% clang++ -O1 -fsanitize=address a.cc ; ./a.out
==10538== ERROR: AddressSanitizer global-buffer-overflow
READ of size 4 at 0x000000415354 thread T0
    #0 0x402481 in main a.cc:3
    #1 0x7f0a1c295c4d in __libc_start_main ??:0
    #2 0x402379 in _start ??:0
0x000000415354 is located 4 bytes to the right of global
  variable 'global_array' (0x4151c0) of size 400
```

ASan report example: use-after-free

```
int main(int argc, char **argv) {
   int *array = new int[100];
   delete [] array;
   return array[argc]; } // BOOM
% clang++ -O1 -fsanitize=address a.cc && ./a.out
==30226== ERROR: AddressSanitizer heap-use-after-free
READ of size 4 at 0x7faa07fce084 thread T0
   #0 0x40433c in main a.cc:4
0x7faa07fce084 is located 4 bytes inside of 400-byte region
freed by thread T0 here:
   #0 0x4058fd in operator delete[](void*) _asan_rtl_
   #1 0x404303 in main a.cc:3
previously allocated by thread T0 here:
   #0 0x405579 in operator new[](unsigned long) _asan_rtl_
   #1 0x4042f3 in main a.cc:2
```

ASan report example: stack-use-after-return

```
int *g;                          int main() {
void LeakLocal() {                 LeakLocal();
  int local;                       return *g;
  g = &local;                    }
}
% clang -g -fsanitize=address a.cc
% ASAN_OPTIONS=detect_stack_use_after_return=1 ./a.out
==19177==ERROR: AddressSanitizer: stack-use-after-return
READ of size 4 at 0x7f473d0000a0 thread T0
    #0 0x461ccf in main   a.cc:8

Address is located in stack of thread T0 at offset 32 in frame
    #0 0x461a5f in LeakLocal()  a.cc:2
  This frame has 1 object(s):
    [32, 36) 'local' <== Memory access at offset 32
```

TSan
(ThreadSanitizer)

Bugs related to concurrency

TSan report example: data race

```
int X;
std::thread t([&]{X = 42;});
X = 43;
t.join();
```

```
% clang -fsanitize=thread -g race.cc && ./a.out
WARNING: ThreadSanitizer: data race (pid=25493)
  Write of size 4 at 0x7fff7f10e338 by thread T1:
    #0 main::$_0::operator()() const race.cc:4     ...
  Previous write of size 4 at 0x7...8 by main thread:
    #0 main race.cc:5
```

MSan
(MemorySanitizer)

Bugs related to contents of memory

MSan report example

```
int main(int argc, char **argv) {
  int x[10];
  x[0] = 1;
  return x[argc];  }
% clang -fsanitize=memory a.c -g; ./a.out
WARNING: Use of uninitialized value
    #0 0x7f1c31f16d10 in main a.cc:4
Uninitialized value was created by an
allocation of 'x' in the stack frame of
function 'main'
```

UBSan
(UndefinedBehaviorSanitizer)

Many other kinds of undefined behavior

UBSan report example: int overflow

```
int main(int argc, char **argv) {
  int t = argc << 16;
  return t * t;
}
% clang -fsanitize=undefined a.cc -g; ./a.out
a.cc:3:12: runtime error:
signed integer overflow: 65536 * 65536
cannot be represented in type 'int'
```

Sanitizers have found thousands of bugs everywhere

Proof links: [1], [2], [3]

But Sanitizers are not enough

- ASan, TSan, MSan, UBSan are “best-effort tools”:
  - Only as good as the tests are
  - Do not prove correctness

- Beyond Sanitizers:
  - Improve test quality (aka test coverage) by fuzzing
  - Protect from security-sensitive bugs in production (hardening)

Fuzzing

What’s “Fuzzing”?

https://en.wikipedia.org/wiki/**Fuzz_testing**

Fuzz testing or fuzzing is a software **testing** technique, often automated or semi-automated, that involves providing invalid, unexpected, or random data to the inputs of a computer program.

Generation-based Fuzzing

- Generate millions of inputs, feed them to the target app
  - Can (and should) be used with Sanitizers
  - May generate invalid inputs (stresses the parser)
  - Or may produce valid inputs by design (e.g. “csmith” -- C fuzzer)
  - Actively used by Chromium security team, found thousands of bugs

- Extremely effective, yet often barely scratches the surface

Mutation-based Fuzzing

- Acquire a test corpus (e.g. crawl the web)
  - Minimize the corpus according to, e.g. code_coverage/execution_time
- Mutate tests from the corpus and run them
- Often better results compared to generation-based fuzzing
  - But harder for highly structured inputs, e.g. C++

Control-flow-guided (coverage-guided) fuzzing

- Same as mutation-based fuzzing, but also
  - Run the mutations with code coverage instrumentation
  - Add the mutations to the corpus if new coverage is discovered
- 1-3 orders of magnitude faster than plain mutation-based fuzzing

AFL-fuzz

AFL-fuzz, a control-flow guided fuzzer

- Instrument the binary at compile-time
  - Regular mode: instrument assembly
  - Recent addition: LLVM compiler instrumentation mode
- Provide 64K counters representing all edges in the app
  - 8 bits per edge (# of executions: 1, 2, 3, 4-7, 8-15, 16-31, 32-127, 128+)
  - Imprecise (edges may collide) but very efficient
- AFL-fuzz is the driver process, the target app runs as separate process(es)

AFL-fuzz is not a toy!

IJG jpeg 1 libjpeg-turbo 1 2 libpng 1 libtiff 1 2 3 4 5 mozjpeg 1 PHP 1 2 3 4 Mozilla Firefox 1 2 3 4 Internet Explorer 1 2 3 4 Apple Safari (1) (2) Adobe Flash / PCRE 1 2 sqlite 1 2 3 4... OpenSSL 1 2 3 4 LibreOffice 1 2 3 4 poppler 1 freetype 1 2 GnuTLS 1 GnuPG 1 2 3 4 OpenSSH 1 2 3 bash (post-Shellshock) 1 2 tcpdump 1 2 3 4 5 6 7 8 JavaScriptCore 1 2 3 4 pdfium 1 2 ffmpeg 1 2 3 4 libmatroska 1 libarchive 1 2 3 4 5 6 ...wireshark 1 2 ImageMagick 1 2 3 4 5 6 7 8 ...lcms 1 libbpg (1) lame 1 FLAC audio library 1 2 libsndfile 1 2 3 less / lesspipe 1 2 3 strings (+ related tools) 1 2 3 4 5 6 7 file 1 2 3 4 dpkg 1 rcs 1 systemd-resolved 1 2 libyaml 1 Info-Zip unzip 1 2 libtasn1 1 2 OpenBSD pfctl 1 NetBSD bpf 1 man & mandoc 1 2 3 4 5 ...IDA Pro [reported by authors] clamav 1 2 3 4 5 libxml2 1 glibc 1 clang / llvm 1 2 3 4 5 6 7 8 ...nasm 1 2 ctags 1 mutt 1 procmail 1 fontconfig 1 pdksh 1 2 Qt 1 wavpack 1 redis / lua-cmsgpack 1 taglib 1 2 3 privoxy 1 perl 1 2 3 4 5 6 7... libxmpradare2 1 2 SleuthKit 1 fwknop [reported by author] X.Org 1 exifprobe 1 jhead [?] capnproto 1 Xerces-C 1 metacam 1 djvulibre 1 exiv 1 Linux btrfs 1 2 3 4 Knot DNS 1 curl 1 2 wpa_supplicant 1 libde265 [] dnsmasq 1 imlib2 1 libraw 1 libwmf 1 uudecode 1 MuPDF 1 libbson 1 libsass 1

LLVM libFuzzer

Sanitizer Coverage instrumentation

- Clang/LLVM flag `-fsanitize-coverage=`
  - `func/bb/edge`: records if a function, basic block or edge was executed
  - `indirect-calls`: records unique indirect caller-callee pairs
  - `8bit-counters`: similar to AFL, provides 8-state counter for edges
    - (1, 2, 3, 4-7, 8-15, 16-31, 32-127, 128+)

- Provides the status in-process and dumps on disk at exit
  - i.e. supports in-process and out-of-process clients
- Should be combined with ASan, MSan, or UBSan
- Typical slowdown: within 10%
  - 8bit counters may be unfriendly to multi-threaded apps

LLVM libFuzzer

- Lightweight in-process control-flow guided fuzzer
  - Provide your own target function
    - `void LLVMFuzzerTestOneInput (const uint8_t *Data, size_t Size);`
  - `-fsanitize-coverage=edge[,indirect-calls][,8bit-counters]`
  - `-fsanitize={address,memory,undefined,leak}`
  - Link with libFuzzer
- Younger than AFL-fuzz and is not as algorithmically sophisticated. Yet quite capable!
- Targeted at libraries/APIs, not at large applications

libFuzzer usage

- Acquire a test corpus, put it into a directory `CORPUS`
  - empty corpus is OK
- Run `./my-fuzzer CORPUS`
  - `-jobs=N`: N parallel jobs, all working on the same corpus
  - `-max_len=N`: limit the input side (default: 64)
  - `-help`: more knobs
- Newly discovered test inputs are written to `CORPUS`
- Bug/timeout will stop the process & dump input on disk
- Optional: feed the produced corpus to AFL-fuzz

Example: FreeType (font rendering library) fuzzer

```
void TestOneInput(const uint8_t *data, size_t size) {
  FT_Face face;
  if (size < 1) return;
  if (!FT_New_Memory_Face(library, data, size, 0, &face)) {
    FT_Done_Face(face);
  }
}
```

Results with FreeType (ASan+UBsan): 45+ bugs

#45999 **left shift of negative value -4592**
#45989 **leak** in t42_parse_charstrings
#45987 512 byte input **consumes 1.7Gb** / 2 sec to process
#45986 **leak** in ps_parser_load_field
#45985 **signed integer overflow**: -35475362522895417 * -8256 cannot be represented in t
#45984 **signed integer overflow**: 2 * 1279919630 cannot be represented in type 'int
#45983 runtime error: **left shift of negative value -9616**
#45966 **leaks** in parse_encoding, parse_blend_design_map, t42_parse_encoding
#45965 left shift of 184 by 24 places cannot be represented in type 'int'
#45964 **signed integer overflow**: 6764195537992704 * 7200 cannot be represented in type
#45961 FT_New_Memory_Face consumes 6Gb+
#45955 **buffer overflow** in T1_Get_Private_Dict/strncmp
#45938 **shift exponent 2816 is too large** for 64-bit type 'FT_ULong'
#45937 **memory leak** in FT_New_Memory_Face/FT_Stream_OpenGzip
#45923 **buffer overflow** in T1_Get_Private_Dict while doing FT_New_Memory_Face
#45922 **buffer overflow** in skip_comment while doing FT_New_Memory_Face
#45920 FT_New_Memory_Face **takes infinite time** (in PS_Conv_Strtol)
#45919 FT_New_Memory_Face **consumes 17Gb** on a small input

Example: OpenSSL

```
SSL_CTX *sctx;
int Init() { ... }
extern "C" void LLVMFuzzerTestOneInput(unsigned char * Data, size_t Size) {
  static int unused = Init();
  SSL *server = SSL_new(sctx);
  BIO *sinbio = BIO_new(BIO_s_mem());
  BIO *soutbio = BIO_new(BIO_s_mem());
  SSL_set_bio(server, sinbio, soutbio);
  SSL_set_accept_state(server);
  BIO_write(sinbio, Data, Size);
  SSL_do_handshake(server);
  SSL_free(server);
}
```

Demo: OpenSSL, the “HeartBleed” bug

Exact commands: llvm.org/docs/LibFuzzer.html

libFuzzer for LLVM itself

- Public bot for
  - Clang: bug 23057
  - Clang-format: bug 23052
  - llvm-as: bug 24639
- Also:
  - libc++ (regex): bug 24411
  - llvm-mc
- Found bugs need to be fixed!

Fuzzing-as-a-service

- Goal: help the developers of core open-source libraries to fuzz their code **continuously**
- Pilot:
  - Regular expressions: PCRE2, RE2
  - Fonts: FreeType, HarfBuzz
  - More to go

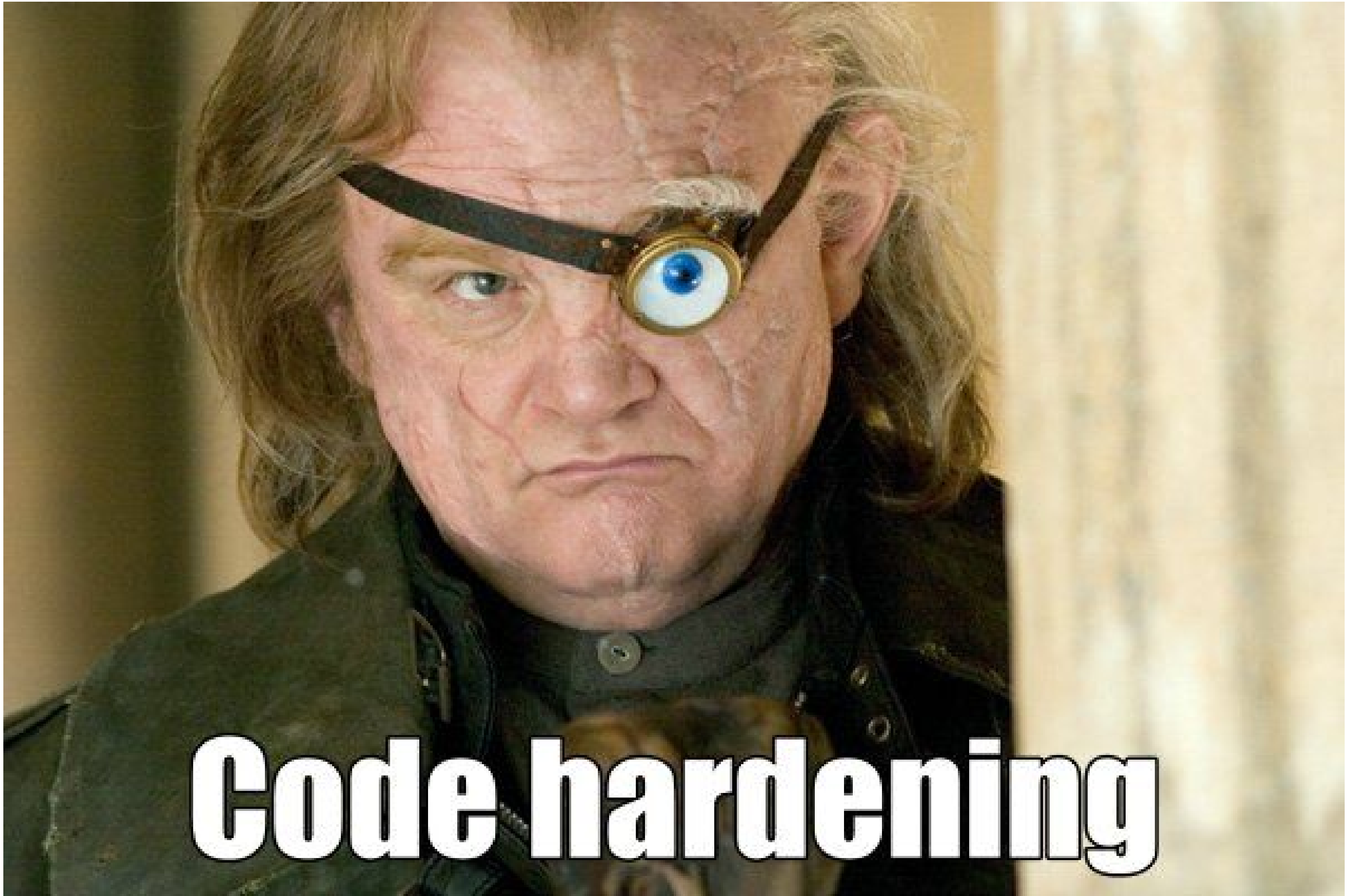
Control-flow-guided fuzzing is not the end

Concolic execution (rocket science)

- Concolic: concrete and symbolic
  - Execute with instrumentation
  - Figure out which branches are never taken
  - Feed the data to SMT solver, get new test inputs that cover more branches
- Good in theory and often in practice too, but very heavyweight

Data-flow-guided fuzzing

- Intercept the data flow, analyze the inputs of comparisons
- Modify the test inputs, observe the effect on comparisons
- Prototype in LLVM libFuzzer (and go-fuzz)
    - Already have trophies (DEMO)
    - May use taint analysis, e.g. DFSan, to make smarter mutations


Code hardening

Threat #1

Buffer-overflow/use-after-free overwrites a vptr or a function pointer by an attacker-controlled value

Hijacked VPTR in Chromium: Pwn2Own 2013 (CVE-2013-0912)

Solution:

Control Flow Integrity (CFI)

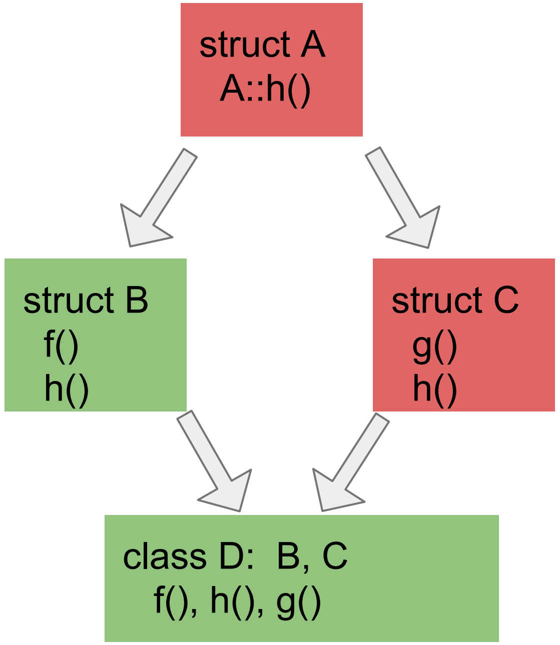
```
clang++ -fsanitize=cfi-vcall -flto
```

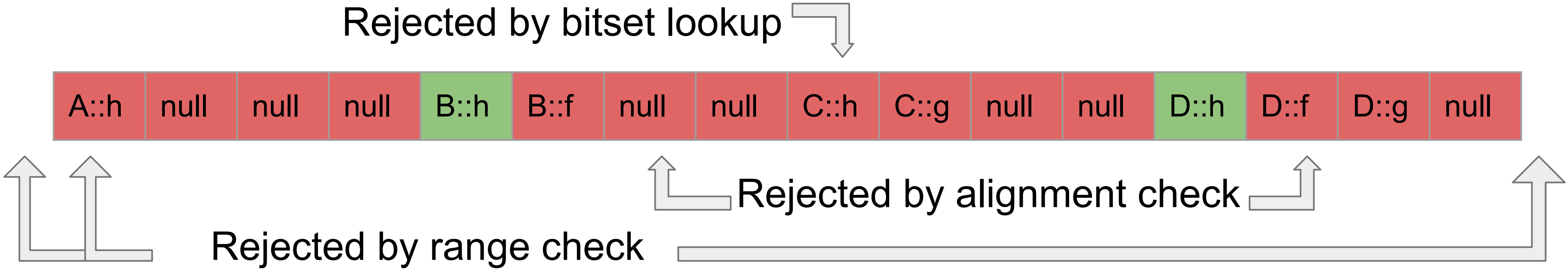
CFI in Clang/LLVM

- Every disjoint class hierarchy is handled separately
  - Assumes the class hierarchy is a closed system; ok for Chrome
- Layout vtables for every hierarchy as a contiguous array
  - Align every vtable by the same power-of-2
- For every virtual function call site
  - Compile-time: compute the strict set of allowed functions
  - Run-time: perform a range check, alignment check, and a bitset lookup

VPTR Layout example (simplified)

```
B *b = ...
b->f(); // Check VPTR
```





Bitset lookup optimizations

- A bitset of <= 64 bits requires no memory loads
- No check if the bitset contains all ones
- Optimize the vtable layouts to minimize the bitset sizes

CFI: generated x86_64 assembler

All ones

```
mov     $0x4008f0,%ecx
mov     %rax,%rdx
sub     %rcx,%rdx
rol     $0x3b,%rdx
cmp     $0x2,%rdx
jae     CRASH
mov     %rbx,%rdi
callq   *(%rax)
…
CRASH:  ud2
```

<= 64 bits

```
mov     $0x400e20,%edx
mov     %rax,%rcx
sub     %rdx,%rcx
rol     $0x3b,%rcx
cmp     $0xe,%rcx
ja      CRASH
mov     $0x4007,%edx
bt      %ecx,%edx
jae     CRASH
mov     %rbx,%rdi
callq   *(%rax)
…
CRASH:  ud2
```

Full check

```
mov     $0x401810,%edx
mov     %rax,%rcx
sub     %rdx,%rcx
rol     $0x3b,%rcx
cmp     $0x40,%rcx
ja      400936 CRASH
testb   $0x1,0x402140(%rcx)
je      400936 CRASH
mov     %rbx,%rdi
callq   *(%rax)
…
CRASH:  ud2
```

More CFI

- Other calls
  - non-virtual member calls: `-fsanitize=cfi-nvcall`
  - C-style indirect calls: `-fsanitize=cfi-icall`

- Casts for polymorphic types
  - Base class => derived class: `-fsanitize=cfi-derived-cast`
  - void * => pointer to a class: `-fsanitize=cfi-unrelated-cast`

CFI & Chromium

- Builds and runs on Linux & Android
  - `...=cfi-vcall,cfi-derived-cast,cfi-unrelated-cast`
  - OSX and Windows are close to working too

- < 1% CPU overhead

- ~7% code size increase

- Significant cleanup was required (real bugs)

Better/different CFI

- Do not require LTO?
  - Requiring LTO is not necessary bad thing!

- Allow class hierarchies to cross the DSO boundaries
  - VS2015 Control Flow Guard (`/d2guard4 + /Guard:cf`)
  - Maybe not a great idea?

Threat #2

Stack-buffer-overflow
overwrites return address
by an attacker-controlled value

Solution:

SafeStack

```
clang++ -fsanitize=safe-stack
```

SafeStack

- Place local variables on a separately mmaped region

- stack-buffer-overflow can’t touch the return addresses

- VPTRs and function pointers can still be affected
  - Combine with CFI

- Chromium: < 1% CPU

SafeStack: code example

```
int main() {
  int local_var = 0x123456;
  bar(&local_var);
}
```

```
push    %r14
push    %rbx
push    %rax
mov     0x207d0d(%rip),%r14
mov     %fs:(%r14),%rbx   # Get unsafe_stack_ptr
lea     -0x10(%rbx),%rax # Update unsafe_stack_ptr
mov     %rax,%fs:(%r14)   # Store unsafe_stack_ptr
lea     -0x4(%rbx),%rdi
movl    $0x123456,-0x4(%rbx)
callq   40f2c0 <_Z3barPi>
mov     %rbx,%fs:(%r14)   # Restore unsafe_stack_ptr
xor     %eax,%eax
add     $0x8,%rsp
pop     %rbx
pop     %r14
retq
```

Summary

- ~~Rely on traditional testing to get a false sense of security~~
- Test with Sanitizers to achieve basic code sanity
  - ASan, TSan, MSan, UBSan
- Use guided fuzzing for stronger security & reliability
  - LLVM libFuzzer and AFL-fuzz make it super easy
- Harden your code for even better security
  - CFI for virtual calls, non-virtual member calls, casts, indirect calls
  - SafeStack for return addresses

Q&A

llvm.org/docs/**LibFuzzer**.html

clang.llvm.org/docs/**ControlFlowIntegrity**.html

clang.llvm.org/docs/**SafeStack**.html