Hardening: current status and trends

Authors

Roman Rusyaev

- Compiler developer
- Compiler team leader

Yuri Gribov (yugr, the_real_yugr), https://github.com/yugr

- System software developer and enthusiast (compilers, runtimes, verification tools, etc.)

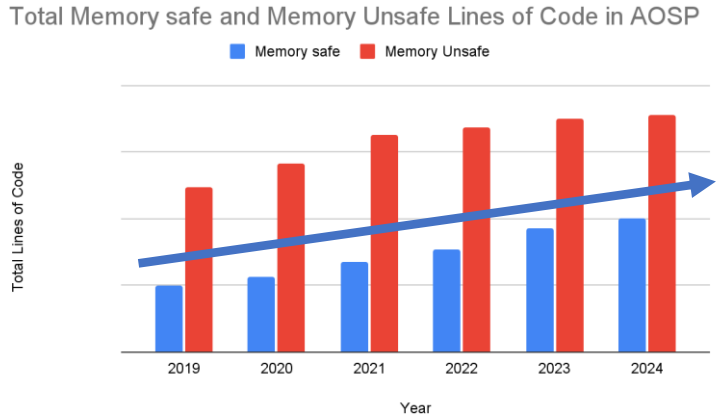
What is hardening

- Tools to detect and/or prevent various types of vulnerabilities
  - Reduction of attack surface
  - E.g. buffer overflow, invalid address access, integer overflow, etc.
- Can be used in release version of product

Mario Simoes, https://snl.no/Mount_Everest

Relevance

- Why safety and stability is relevant for C/C++:
  - 70% of vulnerabilities in Microsoft products are memory errors
    - MSRC Blog: A proactive approach to more secure code (2019)
  - 70% of high/critical bugs in Chromium are memory errors
    - Chromium Security: Memory Safety (2025)
  - 70% of memory errors are 0-day vulnerabilities
    - Improving Memory Safety without a Trillion Dollars (2024)
  - 75% of 0-days are memory errors
    - Safer with Google: Advancing Memory Safety (2024)
  - Memory errors are the most dangerous vulnerabilities
    - Mitre CWE Top 25 2024 (2, 6, 8, 20, 21 places)
  - Adoption of new memory safe languages
  - Customers and governments in many countries *suggest* to use memory safe languages
    - The case for memory safe roadmaps (US CISA, NSA, FBI)



Google Security Blog

Relevance

- Safer with Google: Advancing Memory Safety (2024)
  - The first pillar of our strategy is centered on further increasing the adoption of memory-safe languages
  - While we won't make C and C++ memory safe, we are eliminating sub-classes of vulnerabilities in the code we own
- The Case for Memory Safe Roadmaps (2023)
  - Authoring agencies urge senior executives at every software manufacturer to reduce customer risk by prioritizing design and development practices that implement Memory Safe Languages

Goal of our talk

- Detailed overview of hardening
  - Existing tools
  - How to use them
  - What overhead to expect
- Integration of hardening into C++

https://itoldya420.getarchive.net/amp/media/dart-board-darts-target-sports-6aa47e

Intro

What is Hardening

- Rules to safely
  - Develop SW (secure development process)
  - Deploy SW
    - E.g. symbol stripping, hidden visibility
  - Execute SW
    - Protection in compiler, libraries, OS

Secure development rules

- Concept of Secure Development Lifecycle (e.g. Safe Coding)
- Safe alternatives for dangerous library functions
  - Annex K (memset_s et al.) and other extensions
  - Can be easily faked if not controlled !
- Restrict usage of unsafe functions
  - rand, strcpy, etc.
- Static analysis
  - Mandatory `-Wall -Wextra -Werror`
  - Suggested `-Wformat=2 -Wconversion -Wsign-conversion`
  - Tools: Clang Static Analyzer, Clang-Tidy, etc.
- Specifications (Design-by-Contract)
  - Asserts and C++ contracts

What is Hardening

- Hardening is an integral activity
- In *this* talk we consider only runtime checks
  - Compiler, library, OS
  - Mitigation, not prevention

Linux Hardening Guide

Last edited: March 19th, 2022

Linux is not a secure operating system. However, there are steps you can take to improve it. This guide aims to explain how to harden Linux as much as possible for security and privacy. This guide attempts to be distribution-agnostic and is not tied to any specific one.

Contents

Hardening requirements

- Minimal overhead (2-3% max)
- High precision (no false positives)
- Easy integration
  - ABI compatibility
  - Easy-to-use

Detecting vulnerabilities at QA stage

- Hardening is the last resort – it's much better to detect vulnerabilities at QA
  - QA tools are more powerful and can detect more issues
- Fuzzing, concolic testing, property-based testing, etc.
- AddressSanitizer (≥ 2x overhead)
  - Stack/heap/static overflow, double free, use-after-free/return, etc.
  - State-of-the-art
  - Can be sporadically used in production code for A/B testing
    - Especially low-overhead variants like GWP-Asan and HWASan
- STL debug checks (≥ 2x overhead)
  - `-D_GLIBCXX_DEBUG` in libstdc++ and `-D_LIBCPP_ABI_BOUNDED_ITERATORS` in libc++
  - Change ABI => need full rebuild of all dependencies
- Valgrind (20-50x overhead)
  - Only heap errors (heap overflow, double free, use-after-free, etc.)
  - Much slower than Asan but can sometimes find more errors
- Other tools
  - ElectricFence (only heap overflows), DirtyFrame, etc.

Buffer overflow vulnerabilities

Buffer overflow

- Morris Worm (1988)
- Writing excessive amount of data into program variable

```
char local_buf[32];
sprintf(buf, "Message from user: %s", received_data);
```

- Stack overflow
  - Stack Smashing, Return-to-libc, Return-Oriented Programming attacks
- Heap overflow

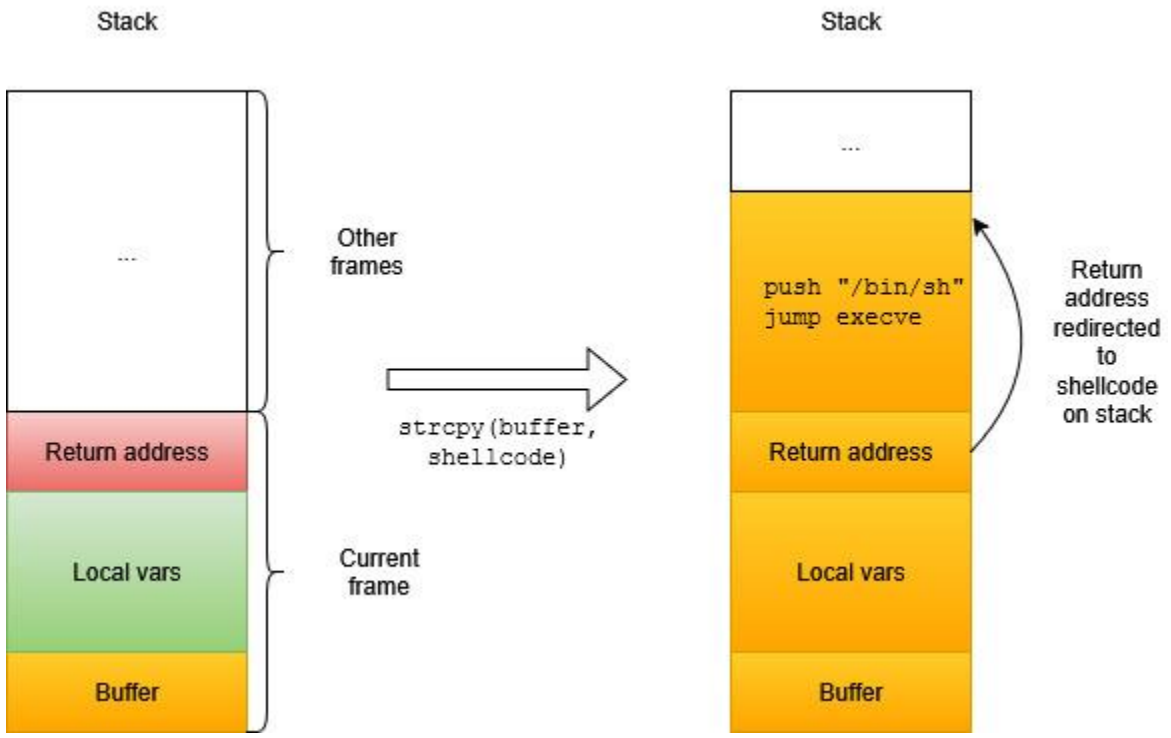
https://www.rawpixel.com/image/5958324/free-public-domain-cc0-photo

Prevalence of buffer overflows

- One of the most dangerous vulnerabilities
  - Mitre CWE Top 25 2024 (2, 6, 20 places)
- 70% vulnerabilities in Microsoft products and Chromium caused by buffer overflows
- 40% of memory-error-based attacks exploit buffer overflows
  - Google Project Zero(2024)
- 80% of Memory Safety CVEs and 46% of Memory Safety KEVs in 2024
  - 20%+ are stack overflows (most dangerous vulnerability)

Stack attacks: Stack Smashing

- Smashing The Stack For Fun And Profit (Aleph One, 1996)
- Write malicious code on stack and executes it on return from function
- No longer relevant due to modern protections



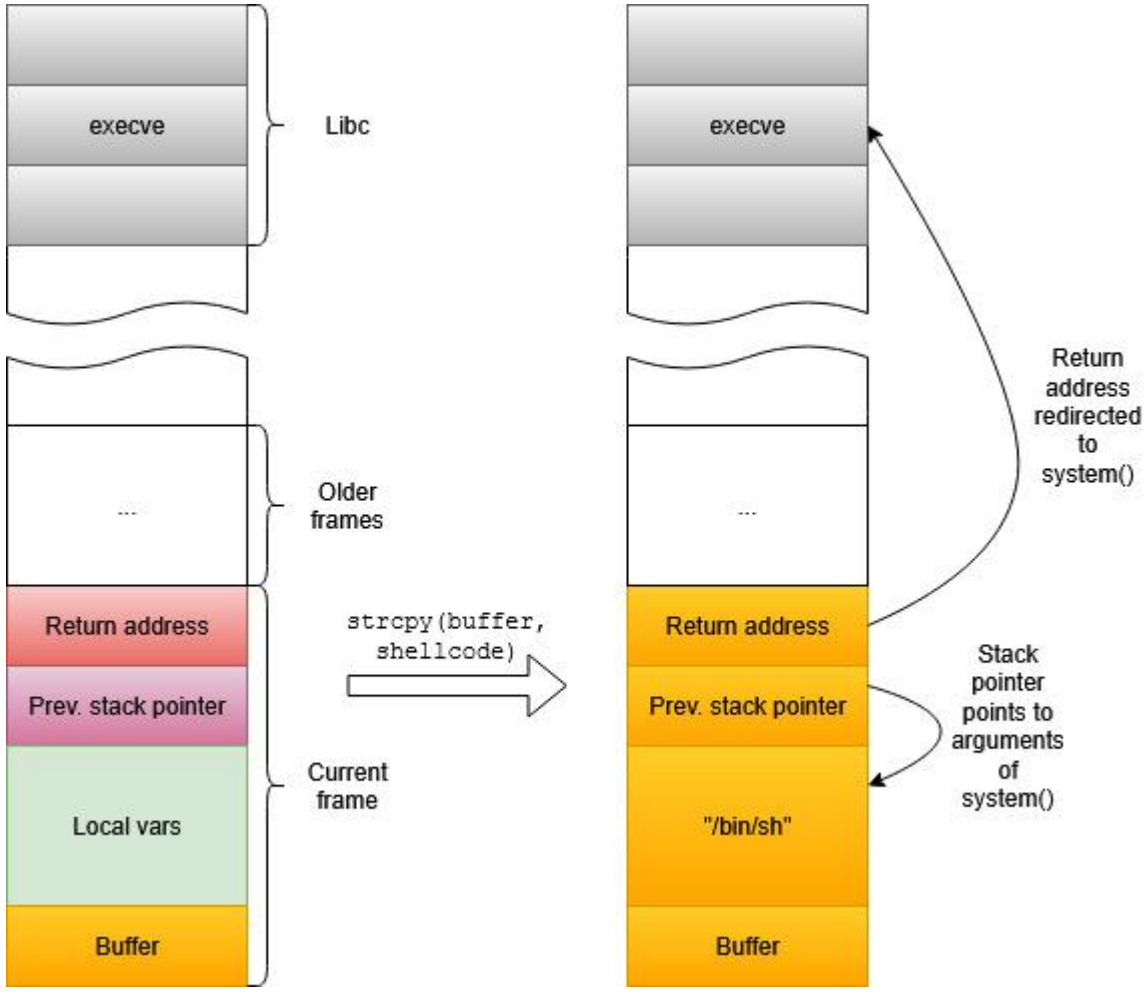
Example: Stack Smashing

```
int main() {
  char buf[32];
  strcpy(buf, code);
}

// Input string from hacker
const char code[] =
  "\x31\xc0"    // xorl %eax, %eax
  "\x50"        // pushl %eax
  "\x68""//sh"  // pushl $0x68732f2f
  "\x68""/bin"  // pushl $0x6e69622f
  ...
  PAD
  // Return address
  "\x0c\xde\xff\xff"
;
```

```
$ CFLAGS='-Wl,-z,execstack -fno-stack-protector -w'
$ PAD=
$ for i in `seq 1 128`; do
  echo PAD=$i
  gcc -m32 -DPAD="\"$PAD\"" -march=i686 $CFLAGS repro.c
  setarch -R env -i ./a.out
  PAD="$PAD\\xff"
done

PAD=1
PAD=2
PAD=3
...
PAD=20
Segmentation fault (core dumped)
PAD=21
Segmentation fault (core dumped)
...
PAD=25
$   # Got shell access
```

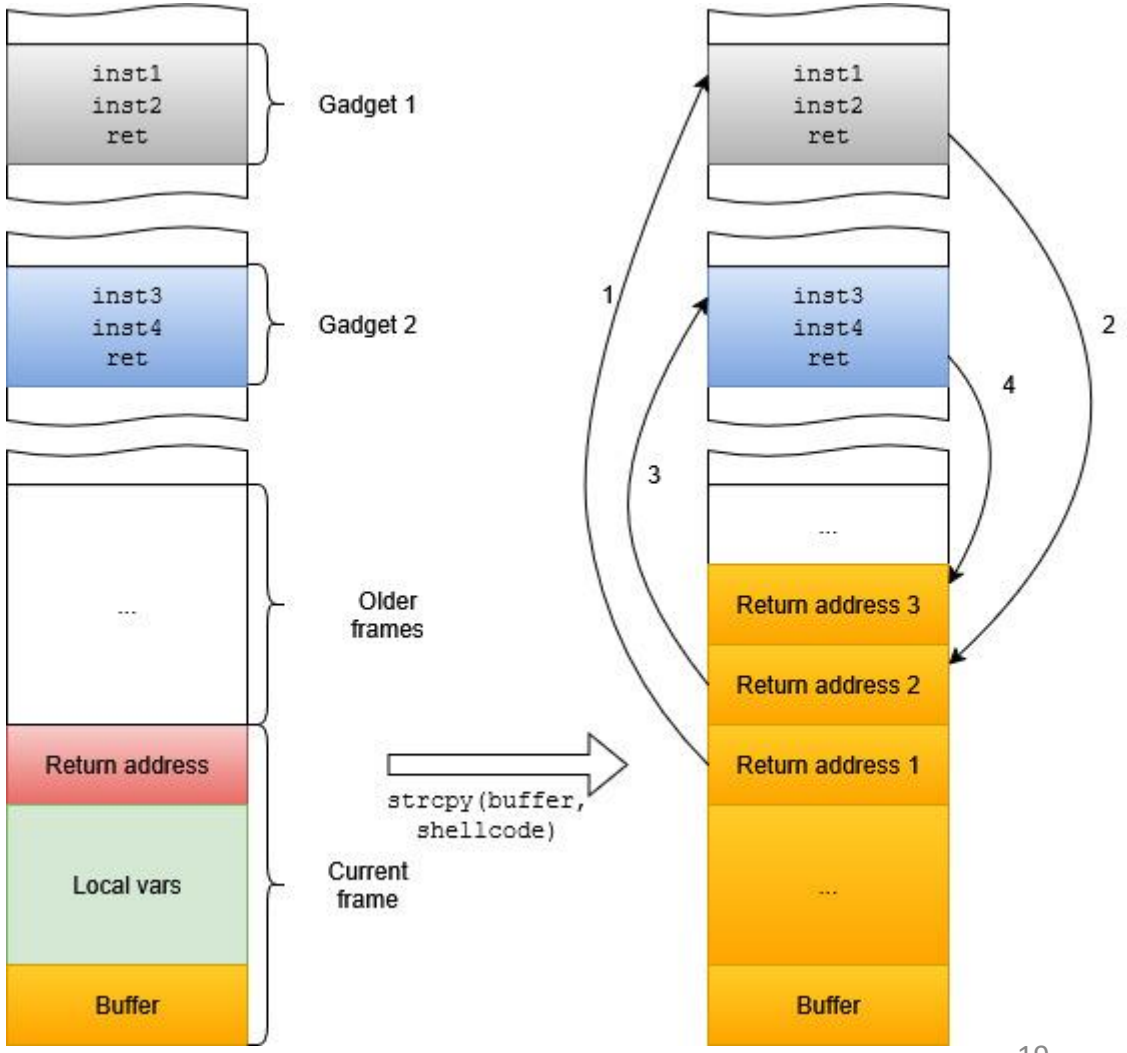
Stack attacks: Return-to-libc

- Solar Designer, 1997
- Jump to standard library API on return from attacked function
  - Usually `execve("bin/sh")` or `system("/bin/sh")`
- Variant of attack: return-to-plt
- Mainly applicable to 32-bit x86 (arguments passed on stack)



Stack attacks: Return-oriented programming

- Nergal, 2001 and Shacham, 2007
- State-of-the-art
- Malicious program constructed from epilogues of normal functions
- Stack contains many return addresses



Heap attacks

https://kingsvilletimes.ca/2022/10/common-sense-health-rake-up-the-leaves-this-fall/

- Based on Heap Overflow vulnerabilities
- More complex and diverse than Stack Overflow
- Main types:
  - Corrupt data in unrelated buffer (e.g. function or vtable pointers)
  - Corrupt allocator metadata
    - E.g. force allocator to write arbitrary address in unrelated malloc/free
    - E.g. corrupt address of malloc hook and call it on next malloc (House of Force attack)

Non-executable data segments

Intro

- W^X / NX bit / Data Execution Prevention
  - Disable execution of code in stack segment
  - Done by OS and startup code
  - Also applied to other writable-segments (heap and globals)
- One of the first hardening defenses
  - First appeared in OpenBSD (2003) and Windows (2004)
  - Fully protects against Stack Smashing attacks
- Enabled by default in all modern Linux distros and on Windows

Issues

- All program code needs to support noexecstack
  - Including linked shared libraries
    - But not dynamically loaded (via dlopen) shared libraries (BZ #32653)
  - Linker will issue warning if noexecstack can't be enabled
    - Suggest to use `LDFLAGS += -Wl,--fatal-warnings`
  - Main reasons for execstack:
    - Legacy assembly code missing the `.note.GNU-stack` directive
    - Pointers to GNU nested functions (rare!)
- No runtime overhead

Address Space Layout Randomization (and PIE)

Intro

- Address Space Layout Randomization
  - Random placement of main program segments (stack, heap, libraries) by OS
  - Hacker no longer knows which return addresses to use in Stack Overflow attacks
- Greatly complicates buffer overflow attacks (ROP, heap overflow, etc.)
  - E.g. Stack Smashing example crashes with SEGV
- One of the first hardening defenses:
  - PaX patch, 2001
  - Linux, 2005
  - Windows, 2007 (Vista, 32-bit Windows overhead was too high due to DLL architecture)
    - Takes decades to add security ... (Theo de Raadt)

https://picryl.com/media/shipping-containers-cargo-port-industry-craft-9326e8

Position-independent Executable (PIE)

- Required for Full ASLR:
  - Main program needs to be compiled in special PIE mode
  - Generated code will not use absolute addressing
  - Allows OS to load main program at random address
- Turned on by default in Ubuntu/Debian (GCC and Clang) and Windows
  - Not default in all Linux distros (e.g. not in Fedora)
  - Even in Debian some critical programs are built without PIE
    - /usr/bin/python3 (Launchpad #1452115)
  - Suggest to explicitly use `-fPIE -pie` in buildscripts
  - `-fPIE ≈ fPIC + -fno-semantic-interposition + -Bsymbolic`

Overhead

- Overhead on modern CPUs is negligible
  - No slowdown in Clang benchmark
  - 32-bit x86 used to have 20% slowdowns
    - Too much PIE is bad for performance (2012)
- ASLR is incompatible with prelinking
  - In the past Prelink was used to speed up library loading at startup in large applications

False negatives

- Vulnerable to info leakage attacks (e.g. Format string attacks):
  - Only base addresses of program and libraries are randomized
  - Hacker knows relative offsets of code, globals, GOT/PLT tables, etc.
  - If address of one object is known the defense is compromised
  - E.g. fork compromises ASLR (Zygote-process in Android)

False negatives

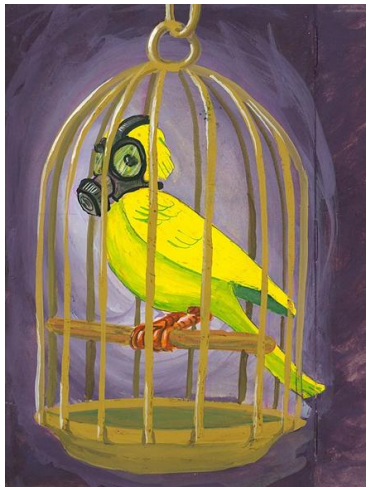
- Not enough randomization
  - Randomization done once (at program start)
  - Only start of mmap addresses is randomized (delta_mmap)
    - Relative offsets of libraries and other mmap regions is fixed
    - Android randomizes order of libraries and inserts random gaps
  - Few bits randomized
    - 16 or even 8 in 32-bit Windows and early Android
  - Not all bits are equally random
  - Windows-specific issues:
    - Every program is randomized once on first execution (for performance)
    - Library loaded at same address in different programs (for performance)
- It’s recommended to regularly restart services to avoid brute-forcing !

ASLR: extensions

- Compile-time randomization
  - Linking object files (or functions) of program in random order
  - Used in OpenBSD (kernel relinked on every boot), Safe Compiler (ISP RAS), etc.
- Runtime diversification:
  - Placing functions in random order at startup
  - Used in Safe Compiler (ISP RAS)
- Moving Target Defense
  - Dynamically relocate functions at runtime

Stack Protector

Stack Protector (Stack Canary)



David & Angie, https://www.flickr.com/photos/studiomiguel/3946174063

- Stack Overflow attacks are based on corruption of return address
- Idea:
  - Place a known random number before return address (stack canary, stack cookie)
  - Prior to return from function check that canary wasn't changed
  - Attacker can't change return address via buffer overflow without also modifying the canary
- One of the first hardening defenses:
  - StackGuard (1997)
  - ProPolice (2001, IBM)
  - StackProtector (2005, RedHat), StackProtectorStrong (2012, Google)
- Greatly reduces risks of all stack overflow attacks (return-to-libc, ROP)
  - E.g. Stack Smashing example fails with

```
*** stack smashing detected ***: terminated
Aborted (core dumped)
```

More safety features

- Scalar variables are placed lower in stack than arrays
  - So that array overflow does not allow hacker to modify scalar flags, function addresses, etc.
- One of canary bytes is always zero (to stop string buffer overflow)

Disadvantages

- Overhead:
  - Need to load canary value, save it on stack, read and check on return
  - 2% on Clang benchmark
  - The Performance Cost of Shadow Stacks and Stack Canaries: 0-9% (2015)
- False negatives:
  - Vulnerable to info leakage: if canary is leaked, protection is compromised
  - Does not protect from corruption of return address by non-overflow attacks
  - Does not protect from heap overflows

Stack split

Intro

- SafeStack, ShadowStack, backward-edge CFI
  - Root cause of stack overflow attacks – return address is stored together with local arrays
  - We can split stack to 2 disjoint parts:
    - Return address (+ maybe scalar variables whose address is never taken)
    - Everything else
- First implemented in StackShield (~2000)
- Comparison with StackProtector:
  - Additional randomization for critical data
  - Protects user pointers to functions on stack
  - StackProtector can still be applied to unsafe part to detect overflows

https://picryl.com/media/reaching-shadow-heart-nature-landscapes-d62bda

Disadvantages

- Overhead:
  - 3% in Clang benchmark
  - 0.1% reported by authors (Clang documentation: SafeStack)
- False negatives:
  - SafeStack currently does not support instrumentation of shared libraries (maybe easy to add: OpenSSF #267)
  - ShadowStack:
    - Supports only AArch64 and RISC-V
    - Protects only return addresses

Stack Clashing (Stack Probes)

Intro

- Stack separated from other segments by unmapped *guard page*
  - Detects stack exhaustion
  - First appeared in Linux (2010)
- Problem:
  - May not detect stack exhaustion for large (>4096 bytes) frames
  - Attacker may corrupt heap or stack of another thread
- Vulnerability reported by Qualys in 2017:
    - The Stack Clash (10 proof-of-concept attacks)
- Idea:
  - In function prologue touch each 4096-th byte of frame (to provoke SEGV)

Overhead

- Overheads are negligible:
  - No slowdown detected in Clang benchmark
  - No performance regressions in Firefox
    - Bringing Stack Clash Protection to Clang / X86 (2021)

Fortification (_FORTIFY_SOURCE)

Example

```
#include <stdlib.h>
#include <string.h>

unsigned n = 4096;

int main() {
  char *a = malloc(1);

  memset(a, 0, n);
  asm("" :: "r"(&a) : "memory");

  a = malloc(200);
  asm("" :: "r"(&a) : "memory");

  return 0;
}
```

```
# No _FORTIFY_SOURCE

$ gcc -U_FORTIFY_SOURCE repro.c -O2 &&
./a.out

Fatal glibc error: malloc.c:2599
(sysmalloc): assertion failed:
(old_top == initial_top (av) &&
old_size == 0) || ((unsigned long)
(old_size) >= MINSIZE && prev_inuse
(old_top) && ((unsigned long) old_end
& (pagesize - 1)) == 0)

Aborted (core dumped)
```

```
# _FORTIFY_SOURCE=3

$ gcc repro.c -O2 && ./a.out

*** buffer overflow detected ***:
terminated

Aborted (core dumped)
```

Implementation

- From Glibc string.h:

```
#if _FORTIFY_SOURCE > 0
__attribute__((always_inline, __nothrow__, leaf))
void *memset (void *__dest, int __ch, size_t __len)
{
  // memset_chk defined in libc.so.6 and performs range check
   return __builtin___memset_chk (__dest, __ch, __len,
                                  __glibc_objsize0 (__dest));
}
#endif
```

- `__glibc_objsize0` calls compiler intrinsic `__builtin_object_size` or `__builtin_dynamic_object_size` (depending on level of protection)
  - `__builtin_object_size` checks pointers to stack variables
  - `__builtin_dynamic_object_size` performs dataflow analysis to also check heap variables

Intro

- Checks valid ranges in C standard library functions (when possible)
  - First appeared in Glibc 2.3.4 (2004)
- ~80 protected functions (exact list of functions in Glibc headers)
  - string.h APIs (memcpy, memset, strcpy, strcat, bzero, bcopy, etc.) – range checks
  - unistd.h APIs (read, pread, readlink, etc.) – range cehcks
- Protects from all types of buffer overflows (stack, heap)
- Implemented via collaboration between
  - Standard library (replaces standard functions with checked versions)
  - Compiler (computes valid ranges)

Overhead

- `-D_FORTIFY_SOURCE=2`: no slowdown in Clang benchmark
- `-D_FORTIFY_SOURCE=3`: 2% in Clang benchmark
- 3% `-D_FORTIFY_SOURCE=2` in ffmpeg (FORTIFY_SOURCE and Its Performance Impact, 2017)

Disadvantages

- Conflicts with Address- and MemorySanitizer:
  - Asan can't analyze XXX_chk functions (sanitizers #247)
  - Combining Asan with fortification causes false negatives (i.e. missed bugs)
    - GCC (not Clang) inserts additional checks for memcpy_chk and memset_chk but it's not enough
  - Fortification is turned on by default in many distros so need to explicitly turn it off in sanitized builds:
    - `-U_FORTIFY_SOURCE` or `-D_FORTIFY_SOURCE=0`
- Supported only in Glibc and Bionic (not in musl or Visual Studio)
  - Standalone implementation: fortify-headers
- Works only under -O and only when standard .h files are used (not implicit declarations)
- Does not check trailing arrays in structs by default (need `-fstrict-flex-arrays`)
- Compiler often can't infer allowed range from context
  - Analysis limited by single function

-fsanitize=bounds

- Fortification approach can be extended to scalar accesses to arrays of known length
- `-fsanitize=bounds` flag in GCC and Clang
  - Analogue of `-D_FORTIFY_SOURCE=2`: checks arrays of known size and VLAs
- Enabled in Android for some critical modules
  - Android Developers Blog: System hardening in Android 11
- No overhead in Clang benchmark
  - Same as `-D_FORTIFY_SOURCE=2`

STL checks

Example

```
#include <stdio.h>
#include <vector>

int main() {
  std::vector<int> v(5, 0);
  asm(“”
    :: "r"(&v)
    : "memory");
  return v[10];
}
```

```
$ g++ tmp.cc
$ ./a.out

$ g++ -
D_GLIBCXX_ASSERTIONS
tmp.cc
$ ./a.out
/usr/include/c++/12/bits/s
tl_vector.h:1123: ... :
Assertion '__n < this-
>size()' failed.
Aborted
```

Intro

- Hardened STL
- List of checks depends on compiler and protection level
  - Index accesses are always checked (also `front`, `back`, etc.) for `std::vector`, `std::deque` and `std::string`
    - Protects from buffer overflow errors
  - GCC:
    - NULL checks in smart pointers (protects from NULL dereference vulnerabilities)
    - Precondition checks (parameters of math. functions, etc.)
  - LLVM:
    - Strict Weak Ordering checks of comparators
      - Painless C++ comparators
  - Visual Studio:
    - Similar checks have very large overheads and will be reimplemented

History and future

- History:
  - First appeared in GCC debug containers as QA check (~2000)
    - Can still (and should) be enabled as `-D_GLIBCXX_DEBUG`
    - ABI-incompatible with normal build
  - `-D_GLIBCXX_ASSERTIONS` option for hardening in GCC (2015)
    - ABI-compatible with normal build
  - Analogous check in libc++ and Safe Buffers proposal (2022)
- In future STL hardening will likely become part of C++ Standard
  - More on this below

Disadvantages

- Overhead:
  - 3.5% in Clang benchmark
  - 0.3% in Google server applications
    - Retrofitting spatial safety to hundreds of millions of lines of C++ (2024)
    - Only with ThinLTO and PGO, otherwise 1-2% (2024)
- False negatives:
  - Detects only subset of overflows (invalid indexing, only STL containers)
  - Other errors are too costly to detect (e.g. iterator errors)

Hardened allocators

Example (1)

```
#include <string.h>
#include <stdlib.h>

void *a, *b;
unsigned n = 4096;

int main() {
  a = malloc(100);
  memset(a, 0xff, n);
  b = malloc(100);
}
```

```
$ gcc -O2 repro.c
$ ./a.out
malloc(): corrupted top size
Aborted
```

Example (2)

```
#include <stdlib.h>

void *a, *b;

int main() {
  a = malloc(1);
  free(a);
  b = malloc(1);
  // Copy-paste error
  free(a);
  return 0;
}
```

```
$ gcc -O2 repro.c

# Glibc does not catch error
$ ./a.out

# But hardened allocator does
$
LD_PRELOAD=libhardened_malloc.
so ./a.out
fatal allocator error: double
free (quarantine)
Aborted
```

Intro

- Additional efforts in heap allocator to complicate exploits and quickly detect heap errors
- Scudo (Android), hardened_malloc, OpenBSD allocator, Glibc MALLOC_CHECK_, etc.
- Protect against various heap errors (heap overflow, double free, use-after-free, free of invalid address)
  - Separate metadata from heap buffers (no “headers")
  - Randomization of addresses in chunks
  - Checksums and/or canaries to detect data/metadata overflows
  - Quarantine (delayed reuse of freed memory)
  - Zeroing of data in free and verifying this in malloc

Disadvantages

- Overhead:
  - 9% in Clang benchmark (hardened_malloc vs Glibc allocator)

GOT protection (Full RELRO)

Intro

- Calls to library functions are done through special trampolines (PLT stubs)
- Trampolines read and update table which holds actual function addresses (GOT)
  - AKA lazy binding
  - Speeds up application startup
- Table kept writable segment where hackers can corrupt it
  - This attack is more rare than buffer overflow
- Solution (read-only relocations, RELRO):
  - Fully initialize table contents at program start and mark it as readonly

Example

```
#include <stdio.h>

void shellcode() {
  printf("You have beeen pwned%s\n", "");
}

extern void *_GLOBAL_OFFSET_TABLE_[];

int main() {
  // Simulate attck
  _GLOBAL_OFFSET_TABLE_[POS] = shellcode;

  puts("Hello world!\n");
  return 0;
}
```

```
$ for i in `seq 0 16`; do
    gcc -Wl,-z,norelro repro.c -
DPOS=$i
    ./a.out
    i=$((i + 1))
  done
  Segmentation fault
  Segmentation fault
  Segmentation fault
  You have beeen pwned
  Hello world!
  Hello world!
  Hello world!
  ...
```

History

- RELRO has been used for initialization of vtables and global dtors (partial RELRO)
  - Ian Lance Taylor: Linker relro
- Needed small adaptations to enable it for GOT (Full RELRO)

Disadvantages

- Negligible overhead
  - No slowdown in Clang benchmark
  - May slow down startup of large programs because all symbols have to be resolved at startup
  - Useful to combine with `-fno-plt` on X86 (up to 10% performance improvement)
- False positives:
  - May break some programs which reference missing symbols in non-executed parts of code
- False negatives:
  - Does not protect user/library function tables (e.g. atexit handlers)

Autoinitialization

Example

```
void foo() {
  char password[32];
  ...
}
void bar() {
  char message[1024];
  if (cond) strcpy(message, "...");
  // Leak password to hacker if !cond
  printf(message);
}
void baz() {
  foo();
  bar();
}
```

Intro

- Forced initialization of local variables
  - Random values for QA run, zeros for hardening
  - Only if compiler failed to prove that they will be always initialized in program
- History:
  - Has been available in commercial toolchains for long time
  - InitAll added to Visual Studio in 2019
    - CppCon 2019: Killing Uninitialized Memory
  - Added to GCC in 2021
    - First discussion in mailing list in 2014
  - Planned for C++26 (P2795, more on this below)
- Prevalence:
  - 10% CVE root cause in Microsoft products in 2018 (from Killing Uninitialized Memory)
  - 12% exploitable bugs in Android (from P2723)

Overheads

- Measurements:
  - 4.5% in Clang benchmark
  - 1% in Firefox (from Trivial Auto Var Init Experiments)
  - Up 10% in hot code (virtio, Chrome)
  - 1-3% on average in Postgres but up to 20% in some scenarios (Ubuntu #1972043)
  - <1% in Windows (Killing Uninitialized Memory)
- Common problem: large local array (e.g. used for IO) on hot path

```
while (std::getline(maps, line)) {
      char modulePath[PATH_MAX + 1] =
"";
      // -ftrivial-auto-var-init
      // will insert memset...
      ret = sscanf(line.c_str(),
                   "%lx-%lx %6s %lx %*s
%*x %" PATH_MAX_STRING(PATH_MAX)
                   "s\n",
                   &start, &end, perm,
&offset, modulePath);
    }
```

Other disadvantages

- Autoinitialization breaks bug detection in Valgrind and Msan
  - Turn it off in QA builds
  - At least compiler warnings still work (`-Wuninitialized -Wmaybe-uninitialized`)
- May rarely enable additional vulnerabilities:
  - Zero initialization (recommended for release builds): “0” is a super-user id in Linux
  - Non-zero initialization: may provoke buffer overflows
- Applied only to local variables
  - Globals are initialized anyway
  - Hardened allocators initialize heap data

Integer overflow checks

Example

```
// From OpenSSH 3.3
nresp = packet_get_int();
if (nresp > 0) {
  // Overflow multiplication to zero ...
  response = xmalloc(nresp*sizeof(char*));
  // ... to cause buffer overflow here
  for (i = 0; i < nresp; i++)
    response[i] = packet_get_string(NULL);
}
```

Intro

- Check integer arithmetic for overflow
  - Default UBsan reports too much debug info so isn't suitable for release builds
  - So use special minimal runtime (with immediate abort)
- Prevalence:
  - Most famous incidents:
    - Therac-25 radiation therapy machine (1985)
    - Ariane 5 rocket crash (1996)
  - ~1% CVE and 1.5% KEV in 2024
  - 23-rd place in Mitre CWE Top 25 2024 (8 in 2019)
- History:
  - `-ftrapv` implemented in GCC in 2000 (patch for -ftrapv option)
    - Wasn't maintained and quickly rotted (e.g. BZ #35412 opened in 2008)
  - Work of John Regehr in 2010
  - Finally UBsan in 2014 (following Asan success)
    - State-of-the-art

Disadvantages

- Overhead:
  - 30% slowdown in Clang benchmark
  - Up to 2x in SPEC (from PartiSan paper, 2018)
  - Disabled by default in Rust due to high overhead
  - We suggest to apply “locally”
- Other issues:
  - UBsan is incompatible with `-fno-strict-overflow` and `-fwrapv`
  - False positives:
    - Isan may report false positives (e.g. need blacklist for code in <random> which relies on unsigned overflow)
  - False negatives:
    - May not detect some bugs which have been “exploited" by optimizer (especially under `-O2`)

Disabling unsafe optimizations

Example error

```
static void
__devexit agnx_pci_remove (struct pci_dev *pdev)
{
  struct ieee80211_hw *dev = pci_get_drvdata(pdev);
  struct agnx_priv *priv = dev->priv;

  // Compiler removes this check
  if (!dev) return;

  ... do stuff using dev ...
}
```

Intro

- Some compilers may too aggressively optimize code that contains easy-to-miss errors and generate unsafe asm
  - Most common problem is dropping user checks
  - Visual Studio is less aggressive than GCC/Clang
- Compiler Introduced Security Bugs
  - Introduced in Silent Bugs Matter: A Study of Compiler-Introduced Security Bugs (2023)
  - Few corresponding CVEs (e.g. CVE-2009-1897) but hundreds of CISBs found in open-source code
- May need to disable such optimizations in code with higher safety requirements

Overhead

- 4.5% slowdown in Clang benchmark
- Small (~1%) overhead in Phoronix Test Suite
  - Performance Impact of Exploiting Undefined Behavior in C/C++ (2025)

How to use ?

- In GCC/Clang disable
  - `-fno-delete-null-pointer-checks`
  - `-fno-strict-overflow` (`== -fwrapv -fwrapv-pointer`)
  - `-fno-strict-aliasing`
- Corresponding bugs can also be detected by UBSanitizer and TypeSanitizer
- Usage:
  - These flags are disabled by default in all compilers/distros
  - But many packages enable at least `-fno-strict-aliasing`
    - Aliasing rules are particularly easy to violate
  - Chrome compiles with all three flags
    - build/config/compiler/BUILD.gn
  - Firefox compiles with `-fno-strict-aliasing`

Control-Flow Integrity

Example

```
include <stdio.h>

struct A { virtual void foo() {} };
struct B : A { void foo() override {} };

struct Evil { virtual void foo() {
  printf("You have been pwned\n"); }
};

A *tmp = new B;

int main() {
  A *a = new A;
  Evil *e = new Evil;
  // Corrupt vtable
  asm("mov %1, %0" : "+r"(a) : "r"(e));
  a->foo();
}
```

```
$ clang++ repro.cc -O2
$ ./a.out
You have been pwned

# CFI detects corruption
$ clang++ -fsanitize=cfi -flto
-fvisibility=hidden repro.cc -
O2
$ ./a.out
Illegal instruction
```

History

- Control Flow Integrity is a generic term for any для violation of original program control flow
  - Including return address corruptions
  - First introduced by Abadi et al. in 2005 (CFI: Principles, Implementations and Applications)
- Two types:
  - forward-edge (call/jump)
  - backward-edge (return)
- A lot of proposed approaches
  - E.g. Stack Protector and Shadow Stack are CFI too
- But nowadays CFI is reserved one of these methods:
  - LLVM CFI, 2015 (2015, Clang 3.7)
  - Microsoft Control Flow Guard, 2014
  - grsecurity RAP, 2016
  - Hardware protections: Intel CET, 2020 (spec 2016) and AArch64 BTI/PAC (2018)

LLVM CFI

- Compiler instrumentation for forward-edge checks
- Implemented only in Clang (not supported by GCC)
- Checks that static and runtime prototype match before calling function by pointer
  - Support vtables and ordinary function pointers
  - (verification algorithms for them are totally different)
- Can also be use for related checks (verify correctness of C++ casts, vtable hijacking, etc.)

Hardware support: Intel CET and AArch64 CFI

- Supported in GCC and Clang
- Much more crude than LLVM CFI
- Check indirect jumps (indirect calls, returns):
  - All valid branch targets are marked with special pseudo-instruction (ENDBR64 on X86)
  - Hardware verifies that target of indirect jump is valid
- Pointer Authentication (AArch64)
  - Top bits of return address hold checksum of return address, frame address and process secret
  - Checksum is, well, checked before return from function

Overhead

- Clang benchmark:
  - No changes under Intel CET
  - 6% overhead under LLVM CFI
- LLVM CFI ~1% in Chrome (Chrome: Control Flow Integrity, 2025)
  - 10% code size increase (so increased I$ and BTB pressure)
- No overhead in Android under LLVM CFI (Android: Security: Control flow integrity, 2025)

Disadvantages

- Fragmentation
  - Three unrelated approaches with different defenses
  - GCC does not support LLVM CFI
- False positives:
  - Many programs need to be updated for LLVM CFI (fix `reinterpret_cast<void *>`, etc.)
  - E.g. vanilla Clang does not pass LLVM CFI (need blacklists)
- False negatives:
  - LLVM CFI:
    - Only type mismatches (hacker can still call wrong function if types match)
    - Harder to integrate (need LTO, issues with cross-DSO type checks)
  - Intel/AArch64: do not check types at all
  - Jump tables not checked (only in CET with `-mcet-switch` which is disabled by default)

Compiler options

Protection	Flags
Noexecstack	On by default
Full ASLR (PIE)	`-fPIE -pie`
Stack Protector	`-fstack-protector-strong`
Safe Stack	`-fsanitize=safe-stack`
Stack Clashing	`-fstack-clash-protection`
_FORTIFY_SOURCE=2	`-D_FORTIFY_SOURCE=2` (or `3`), `-fsanitize=bounds` (also recommend `-fstrict-flex-arrays=1`)
STL hardening	`-D_GLIBCXX_ASSERTIONS` (libstdc++), `-D_LIBCPP_HARDENING_MODE=...` (libc++)
Hardened allocator	`LD_PRELOAD=path/to/allocator.so` `MALLOC_CHECK_=3` or `GLIBC_TUNABLES=glibc.malloc.check=3` (Glibc)
Full RELRO	`-Wl,-z,relro -Wl,-z,now`
Autoinitialization	`-ftrivial-auto-var-init=zero`
Integer Overflow	GCC: `-fsanitize-trap=signed-integer-overflow,pointer-overflow` Clang: `-fsanitize=signed-integer-overflow,pointer-overflow -fsanitize-minimal-runtime` (also recommend `integer`)
Disable optimizations	`-fno-delete-null-pointer-checks` `-fno-strict-overflow` (== `-fwrapv -fwrapv-pointer`) `-fno-strict-aliasing`
Control-flow integrity	LLVM: `-fsanitize=cfi -flto=thin -fvisibility=hidden -fsanitize-cfi-cross-dso` Intel CET: `-fcf-protection` AArch64: `-mbranch-protection=standard` (why not `-fcf-protection` ?!)

Real-world adoption

Linux distros

Protection	Ubuntu 24.04			Debian 12			Fedora 42		
	GCC	Clang	Pkgs	GCC	Clang	Pkgs	GCC	Clang	Pkgs
Noexecstack	Y	Y	Y	Y	Y	Y	Y	Y	Y
Full ASLR (PIE)	Y	Y	Y	Y	Y	Y	N	N	Y
Stack Protector	Y	N	Y	N	N	Y	N	N	Y
Safe Stack	N	N	N	N	N	N	N	N	N
Stack Clashing	Y	N	Y	N	N	N*	N	N	Y
_FORTIFY_SOURCE	Y (2)	N	Y (2)	N	N	Y (2)	N	N	Y (3)
STL hardening	N	N	N	N	N	N	N	N	Y (libstdc++)
Hardened allocator	N	N	N	N	N	N	N	N	N
Full RELRO	Y	N	Y	N	N	Partial	N	N	Y
Autoinitialization	N	N	N	N	N	N	N	N	N
Integer Overflow	N	N	N	N	N	N	N	N	N
Disable optimizations	N	N	N	N	N	N	N	N	N
Hardware CFI (Intel CET, AArch64 BP)	Y	N	Y	N	N	N*	N	N	Y

- Default protections differ across distros
- Clang enables much fewer protections by default
- New protections not enabled by default
- Distro packages are protected better than user programs
- Default distro protections may be disabled even in critical packages (e.g. no PIE/Full RELRO in python3 in Debian 12, only 85% packages in Debian 10 enabled StackProtector)

* Will be enabled in next Debian version

Browsers

Protection	Chrome (140.0.7313.1)	Firefox (142.0b1)
Noexecstack	Y	Y
Full ASLR (PIE)	Y	Y
Stack Protector	Y (weak)	Y
Safe Stack	N	N
Stack Clashing	N	Y
_FORTIFY_SOURCE=2	Y	Y
STL hardening	Y (libstdc++)	N
Hardened allocator	Y	N
Full RELRO	Y	Y
Autoinitialization	Y	N
Integer Overflow	N	N
Disable optimizations	Y	Y
CFI	Y (LLVM on X86, AArch64 CFI)	N

- Considered default flags on Linux

Hardening in memory-safe languages

Hardening in Rust

Protection	Needed ?	Rust status	Notes
Noexecstack	Only unsafe/external code	On	Prooflink
ASLR	Only unsafe/external code	On	Prooflink
Stack Protector	Only unsafe/external code	**Nightly flag**	Prooflink
Safe Stack	Only unsafe/external code	**Nightly flag**	Prooflink
Stack Clashing	Yes	On	Prooflink
_FORTIFY_SOURCE, STL hardening	No	No	Already in lang.
Hardened allocator	Only unsafe/external code	hardened_malloc bindings	Prooflink
Full RELRO	Only unsafe/external code	On	Prooflink
Autoinitialization	No	N/A	Already in lang.
Integer Overflow	Yes	Flag (default off)	
CFI	Only unsafe/external code	**Nightly flag**	Prooflink

- All memory-related Rust CVEs are caused by errors in unsafe code (Xu et al., 2021)
- 50% popular crates have unsafe blocks (Evans et al., 2020)
- stdlib::core 8.5% unsafe, nalgebra 4.5%, rav1e 7%, tokio 5%, veloren 15%
  - (modulo errors in my scripts…)
- Stable Rust lacks flags for some key defenses

Options that we didn’t cover

- Options for cleaning secrets (passwords, keys, etc.):
  - Stack scrubbing – clear stack on return from function (`-fstrub`)
  - Clear registers on return from function (`-fzero-call-used-regs`)
- Options for side-channel attacks (Spectre, etc.)
- `-fhardened` – umbrelly flag for most important hardening protections
  - Good default flag but currently available only in GCC (LLVM #122687)
  - Enables all flags recommended by OpenSSF (see `--help=hardened`)

Anatomy of fortification

How fortification works

- When _FORTIFY_SOURCE macro is enabled, standard functions are implemented as inline stubs which call special compiler builtins
- Builtins expand to
  - Normal function calls
    - If compiler failed to determine size of dst
    - If compiler can prove that size of dst >= size of src
  - *_chk versions which check sizes at runtime

__builtin_object_size

```
size_t __builtin_object_size (const void *ptr, /*detail*/)
```

- Returns
  - Size of object addressed by ptr
  - -1 if size is not known at compile time

```
size_t __builtin_dynamic_object_size(const void *ptr, /*detail*/)
```

- Same as __builtin_object_size but size may also be computed at runtime
  - Needed for malloc, alloca, VLA

Strcpy example

```
int main(int argc, char *argv[]) {
  strcpy(a, argv[1]);
  puts(a);
}
```

```
int main(int argc, char *argv[]) {
  __builtin___strcpy_chk (a, argv[1], __builtin_object_size(a, ...));
  puts(a);
}
```

__builtin_object_size(...) > src_len

__builtin_object_size(...) ≯ src_len

```
extern char a[];

int main(int argc, char *argv[]) {
  strcpy(a, argv[1]);
  puts(a);
}
```

```
char a[4];

int main(int argc, char *argv[]) {
  __strcpy_chk(a, argv[1], dst_len);
  puts(a);
}
```

Strcpy example

```
int main(int argc, char *argv[]) {
  __strcpy_chk (a, argv[1], dst_size);
  puts(a);
}
```

```
char * __strcpy_chk(char *dest, const char *src, size_t destlen){
  size_t len = strlen(src);

  if (len >= destlen)
    abort();

  return memcpy(dest, src, len + 1);
}
```

Side effect in Clang

```
int main() {
  void *p;
  size_t n;
  if (getenv("TEST")) {
    p = &n;
    n = sizeof n;
    puts("Hello");
  } else {
    p = nullptr;
    n = 0;
  }
  memset(p, 0, n);
}
```

With fortification

```
int main() {
  getenv("TEST");
  puts("Hello");
}
```

- With fortification memset argument has `nonnull` attribute
- So compiler will drop the nullptr branch
- More details in LLVM #60389

Support for hardening in language

New behavior type in C++

- Types of behavior in C++
  - Undefined
    - Optimizer can do arbitrary transforms
  - Unspecified
    - Depends on implementation (undocumented)
  - Implementation-defined
    - Depends on implementation (documented)

New behavior type in C++

- Types of behavior in C++
  - Undefined
    - Optimizer can do arbitrary transforms
  - Unspecified
    - Depends on implementation (undocumented)
  - Implementation-defined
    - Depends on implementation (documented)
  - **Erroneous**

Erroneous behavior

- Defined behavior for incorrect code
  - Should not enable vulnerabilities
- Opposite of UB
- C++26 uses erroneous behavior for uninitialized variables (P2795)
  - Maybe also for some missing returns (P2973)
- Most likely will be implemented on top of existing `-ftrivial-auto-var-init` flag

```
int foo(){
  int x;
  return x ? 1 : 1;
}
```

C++23 (UB)

```
int foo(){
  abort();
}
```

```
int foo(){
  while(true);
}
```

```
int foo(){
  return 0x42;
}
```

C++26 (Erroneous)

```
int foo(){
  return 1;
}
```

[[indeterminate]] attribute

- Disables Erroneous Behavior for local variable or function parameter (no initialization, back to UB)
- Used for optimizations

```
void f(int);

void g() {
  int x [[indeterminate]]; // indeterminate value
  int y;                   // erroneous value

  f(x); // undefined behavior
  f(y); // erroneous behavior
}
```

STL hardening in C++ Standard

- P3471
- Preconditions for some functions of standard library
  - Hardened preconditions
    - E.g. `idx < size` in `vector::operator[]`
  - Will be implemented via contracts (precondition assertions)
- Toolchains will provide special flags to enable them
  - E.g. `-fhardened` or `-D_LIBCPP_HARDENING_MODE`

C++ Profiles

- Evolution and standardization of C++ Core Guidelines
- Language dialects
  - Prohibit unsafe or uncheckable constructs
  - Enforced by local static analysis
- Migration tools
  - C++ Safe Buffers, Clang-Tidy fix-its
    - Already used in Chrome
- Candidates for Safety Profiles:
  - STL hardening
  - Prohibit raw pointers
  - RAII
  - Prohibit unsafe casts
  - Etc.
- Will profiles unify all hardening protections in future?

Safety Profiles:
Type-and-resource Safe programming in ISO Standard C++

Bjarne Stroustrup
Columbia University
www.stroustrup.com
Gabriel Dos Reis
Microsoft

Conclusions

Conclusions

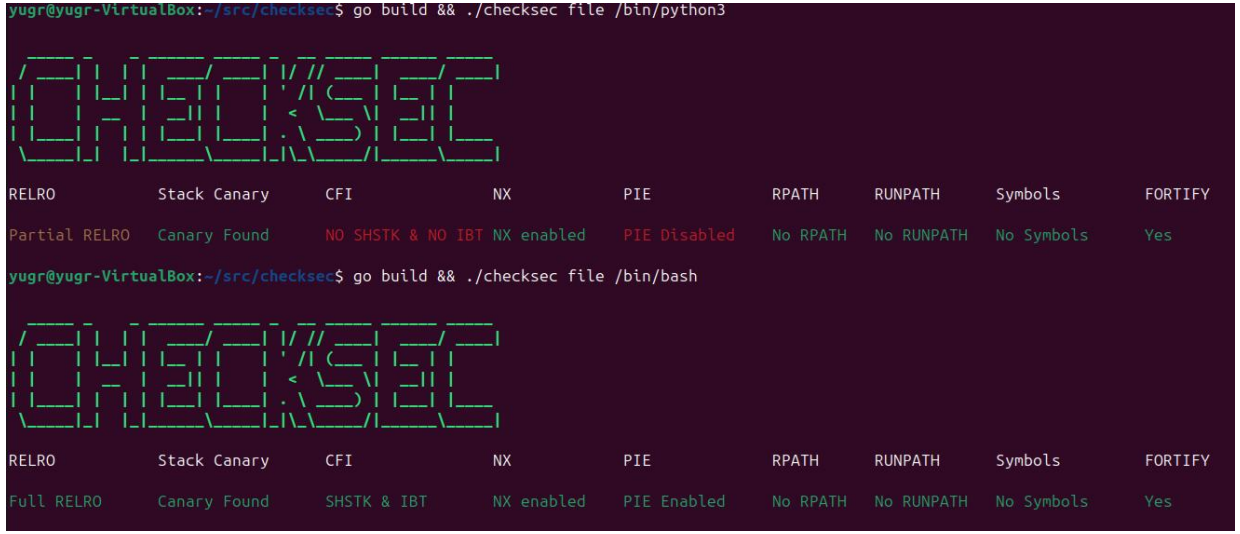
- Hardening is a state-of-the-art part of modern C/C++ development
- There are many methods with different protection levels and overhead
- Will be used more and more due to adoption of memory-safe languages and tightening of government requirements
  - Also standardized in the language
- Will this make C++ more competitive against memory safe languages?

Some suggestions

- Check compile flags used in your release build and for external libraries/programs (e.g. distro packages that your service relies on)
  - Check what hardening flags are enabled by default in your toolchain
- Decide with Security Team what hardening methods to enable in which components
- *Minimal* recommended protection:
  - ASLR: `-fPIE -pie`
  - Stack Protector: `-fstack-protector-strong`
  - Fortification: `-D_FORTIFY_SOURCE=2`
  - Full RELRO: `-Wl,-z,relro -Wl,-z,now`
  - Stack Clash protection: `-fstack-clash-protection`
  - Control-flow Integrity: `-fcf-protection` on X86, `-mbranch-protection` on AArch64

Some suggestions

- Automatically detect under-protected components (no-PIE, etc.) with checksec
  - Need to build it yourself (distros have legacy checksec.sh which doesn't support many defenses)



- Can detect missing noexecstack, PIE, _FORTIFY_SOURCE, RELRO, etc.
  - Supports only dynamically linked apps/libraries
  - Does not (yet ?) support new defenses: Stack Clashing (#300), Safe Stack (#301), LLVM CFI

What we didn’t cover

- Hardening in other popular Linux distros (RedHat/Alma, OpenSUSE, Alpine, etc.) and embedded systems
  - Building Embedded Systems Like It’s 1996 (2022)
- Hardening in other OSes (Android, Windows, macOS, BSDs)
- Hardening in OS kernel
  - Linux Kernel Defence Map (2025)
- Hardening in critical SW
  - Chats, mail clients, multimedia, interpreters, DB, office SW, readers
- Hardening in other memory-safe languages (Java, Swift, Ada, Solidity, etc.)
- Hardening in JIT compilers
- Hardware support for hardening (AArch64 TBI and MTE, CHERI)

Recommended reading

- Hardening guides
  - OpenSSF Compiler Options Hardening Guide for C and C++
  - Linux Hardening Guide
- Overview of attacks
  - Nightmare
  - Overview of GLIBC heap exploitation techniques
- Blogs
  - Google Security Blog
  - Embedded in Academia (John Regehr): A Guide to Undefined Behavior in C and C++ and UB in 2017

Acknowledgements

- Serge Bronnikov (VK Tech/Tarantool)
  - https://bronevichok.ru/
- Roman Lebedev (Spectral::Technologies)
- Organizers of C++ Zero Cost conference

The End

- Latest version
  - https://github.com/yugr/slides/blob/main/CppZeroCost/2025/EN.pptx

- Questions ?

Appendix

Reproducing results

- Distro versions:
  - We checked the latest *stable* versions
  - Debian 12 (bookworm), Fedora 42, Ubuntu 24.04 (noble)
- Browser versions:
  - Firefox: https://github.com/mozilla-firefox/firefox (FIREFOX_142_0b1_RELEASE tag)
  - Chrome: https://chromium.googlesource.com/chromium/src (140.0.7313.1 tag)
- Clang benchmark:
  - Compiled CGBuiltin.cpp with -O2 (the largest file in LLVM codebase)
  - https://github.com/yugr/slides/tree/main/CppZeroCost/2025/bench
- Script to compute CVE/KEV metrics:
  - https://github.com/yugr/slides/tree/main/CppZeroCost/2025/scripts
- Prooflinks and additional info (in Russian):
  - https://github.com/yugr/slides/blob/main/CppZeroCost/2025/plan.md

Stack Protector: How to enable ?

- Turned on by default only in Ubuntu GCC (not in Fedora and Debian, not in Clang) and Windows
  - We suggest to explicitly specify `-fstack-protector-strong`
  - Packages in Debian, Fedora, Ubuntu are built with this flag
    - Only 85% packages in Debian 10 used SP (Building Embedded Systems Like It's 1996)
- Turned on in release build of Firefox
- Weak mode turned on in release build of Chrome

Safe Stack: How to enable ?

- Several implementations:
  - SafeStack: `-fsanitize=safe-stack` (most commonly used)
  - Intel CET Shadow Stack: `-fcf-protection` (most commonly used)
  - ShadowCallStack: `-fsanitize=shadow-call-stack`
- Not turned on by default in distros and Chrome/Firefox

Stack Clashing: How to enable ?

- Turned on by default only in Ubuntu GCC (not in Fedora and Debian, not in Clang)
  - We suggest to explicitly specify `-fstack-clash-protection`
  - Packages in Debian, Fedora, Ubuntu are build with this flag
- Firefox uses Stack Clash (BZ #1852202), Chrome does not

_FORTIFY_SOURCE: How to enable ?

- Can be turned on via `-D_FORTIFY_SOURCE=2` or `-D_FORTIFY_SOURCE=3`
  - Until `-D_FORTIFY_SOURCE=4` is introduced ☺
- Turned on by default in Ubuntu GCC (`-D_FORTIFY_SOURCE=3`)
  - Not in Debian and Fedora, not in Clang
- Real-world adoption
  - Debian packages are compiled with `-D_FORTIFY_SOURCE=2` by default
  - Ubuntu packages with `-D_FORTIFY_SOURCE=3`
  - Fedora packages with `-D_FORTIFY_SOURCE=3` (since 2023)
  - Chrome and Firefox are compiled with `-D_FORTIFY_SOURCE=2`

STL hardening: How to enable ?

- Libstdc++: `-D_GLIBCXX_ASSERTIONS`
  - (default STL in GCC and Clang)
- Libc++: `-D_LIBCPP_HARDENING_MODE=...`
  - (enabled in Clang via `-stdlib=libc++`)
- Visual Studio: `-D_ITERATOR_DEBUG_LEVEL=1`
- Not enabled by default in Debian, Ubuntu and Fedora compilers
- Real-world adoption:
  - Enabled by default for packages in Fedora but not in Debian and Ubuntu
  - Google: Chrome and server systems
    - Retrofitting spatial safety to hundreds of millions of lines of C++

Hardened allocators: How to enable ?

- Usually just `LD_PRELOAD=path/to/allocator.so`
- Additional Glibc checks can be enabled via `MALLOC_CHECK_=3` or `GLIBC_TUNABLES=glibc.malloc.check=3`
- Real-world adoption:
  - Most Linux distros use Glibc
  - Android uses Scudo
  - Chrome uses PartitionAlloc (hardened allocator)
    - Efficient And Safe Allocations Everywhere!
  - Firefox does not use hardened allocator :(
    - Firefox and Chromium: Memory Allocator Hardening

Full RELRO: How to enable ?

- Linker flags for Full RELRO: `-Wl,-z,now -Wl,-z,relro`
  - Enabled by default in Ubuntu GCC but not in Clang (Clang has only partial RELRO)
  - Not enabled by default in Debian and Fedora (neither in GCC, nor in Clang)
- Real-world adoption
  - Ubuntu and Fefora packages are compiled with Full RELRO by default
  - Debian packages do not have Full RELRO by default
  - Enabled in Chrome (BUILD.gn) and Firefox (flags.configure)

Autoinit: How to enable ?

- `-ftrivial-auto-var-init=zero` flag in GCC and Clang
  - Not enabled by default in compilers in Ubuntu, Debian, Fedora
- Hidden flag -initiall in Visual Studio
- Real-world adoption:
  - Not enabled by default for packages in Ubuntu, Debian, Fedora
  - Discussion in Ubuntu tracker ongoing (#1972043)
- Enabled in Chrome (Chromium #40633061)
  - Fixing hot paths took ~4 months
- Not yet enabled in Firefox (Trivial Auto Var Init Experiments)
- Enabled in Android user- and kernelspace (System hardening in Android 11)

Integer overflow checks: How to enable ?

- GCC: `-fsanitize-trap=signed-integer-overflow,pointer-overflow`
  - GCC does not support`integer`
  - Note that `-ftrapv` *does not work*
- Clang: `-fsanitize=signed-integer-overflow,pointer-overflow -fsanitize-minimal-runtime`
  - Suggest to use `-fsanitize=integer` (may need to add some STL headers e.g. <random> to blacklist)
- Real-world adoption:
  - Not used in Ubuntu, Debian, Fedora, not used in Chrome and Firefox
  - Enabled in Android media stack:
    - Android Developers Blog: Hardening media stack
    - Android Developers Blog: Compiler-based security mitigations in Android P

CFI adoption

- Enabled by default in Android
- LLVM CFI not enabled by default for packages in Ubuntu, Debian, Fedora
  - LTO + lack of support in GCC
- Intel CET and AArch64 CFI enabled by default for packages in Ubuntu, Fedora and Debian
- Chrome uses LLVM CFI on X86 and AArch64 CFI on ARM
  - Firefox does not enable any CFI

CFI: How to enable ?

- LLVM CFI:
  - Not enabled by default in compilers in Ubuntu, Debian, Fedora
  - Turn on via `-fsanitize=cfi` (also needs `-flto=thin -fvisibility=hidden`)
  - (LTO for building program call graph, visibility to reduce number of edges)
  - Cross-DSO calls require `-fsanitize-cfi-cross-dso` (hurts performance)
- Intel CET:
  - Turn on via `-fcf-protection`
    - In past also needed `-mcet`, `-mshstk` and `-mibt`
  - On by default in Ubuntu GCC (Toolchain: Compiler flags)
- AArch64 CFI:
  - Turn on via `-mbranch-protection=standard`
  - Noone knows why `-fcf-protection` wasn't reused :(