Vrije Universiteit Amsterdam

SafeInit: Comprehensive and Practical Mitigation of Uninitialized Read Vulnerabilities

Alyssa Milburn, Herbert Bos,
Cristiano Giuffrida

riscure

Challenge your security

Contact: Alyssa Milburn (milburn@riscure.com)

Riscure is hiring! visit www.riscure.com/careers

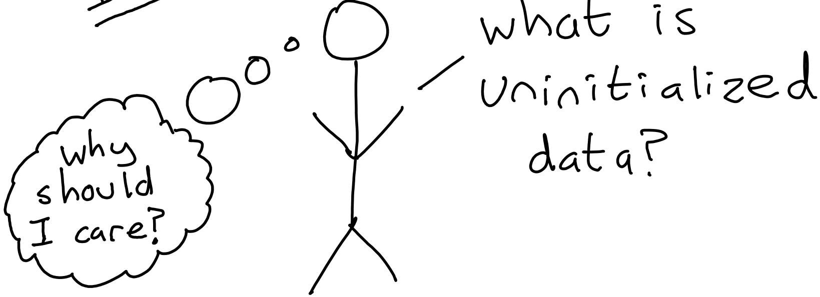
Riscure B.V.
Frontier Building, Delftechpark 49
2628 XJ Delft
The Netherlands
Phone: +31 15 251 40 90

www.riscure.com

Riscure North America
550 Kearny St., Suite 330
San Francisco, CA 94108
USA
Phone: +1 650 646 99 79

inforequest@riscure.com

INTRODUCTION



Information exposure

```
keyMemory = malloc();
*keyMemory = cryptoKeys;
```

Heap allocation is currently used by: **keyMemory**

(Secret) encryption keys

Information exposure

```
keyMemory = malloc();
*keyMemory = cryptoKeys;
free(keyMemory);
```

Heap allocation is currently used by: **nothing**

(Secret) encryption keys

Information exposure

```
keyMemory = malloc();
*keyMemory = cryptoKeys;
free(keyMemory);

buffer = malloc();
```

Heap allocation is currently used by: **buffer**

(Secret) encryption keys

Information exposure

```
keyMemory = malloc();
*keyMemory = cryptoKeys;
free(keyMemory);

buffer = malloc();
send(sock, buffer, ...);
```

Heap allocation is currently used by:

(Secret) encryption keys

Uninitialized read vulnerabilities

Uninitialized read vulnerabilities

- Information exposure
    - Encryption keys, passwords

Uninitialized read vulnerabilities

- Information exposure
  - Encryption keys, passwords
  - Pointers (ASLR)

Uninitialized read vulnerabilities

- Information exposure
  - Encryption keys, passwords
  - Pointers (ASLR)

- Control flow attacks

Uninitialized read vulnerabilities

- Information exposure
    - Encryption keys, passwords
    - Pointers (ASLR)
- Control flow attacks
- Undefined behaviour

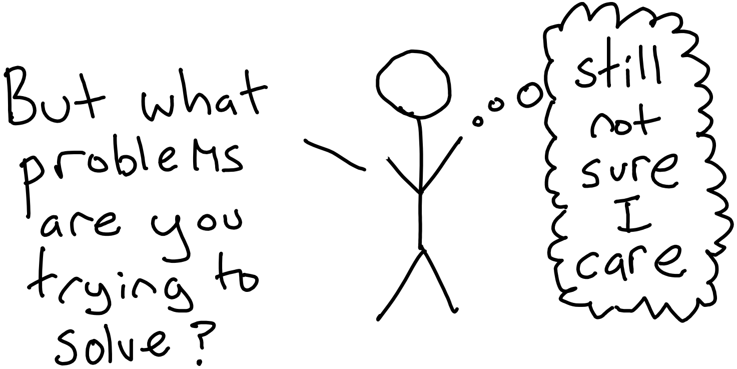
Compiler warnings?

```
warning:
‘variable’ may be used
uninitialized in this function
```

Compiler warnings?

github search for `-Wno-uninitialized`:

118,732 code results


But what problems are you trying to solve?
still not sure I care

qemu

qemu

CVE-2016-4020: i386: leakage of stack memory to guest in kvmvapic.c

qemu

CVE-2016-4020: i386: leakage of stack memory to guest in kvmvapic.c

```
-       uint32_t imm32;
+       uint32_t imm32 = 0;
```

qemu

CVE-2016-4020: i386: leakage of stack memory to guest in kvmvapic.c

```
-     uint32_t imm32;
+     uint32_t imm32 = 0;
```

CVE-2016-5105: scsi: megasas: stack information leakage while reading configuration

qemu

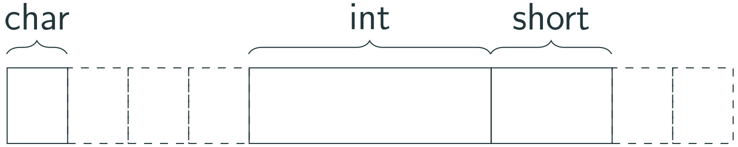
CVE-2016-4020: i386: leakage of stack memory to guest in kvmvapic.c

```
-     uint32_t imm32;
+     uint32_t imm32 = 0;
```

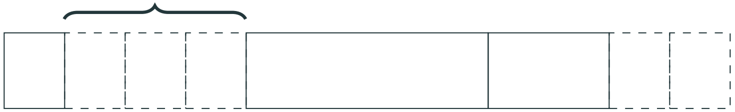
CVE-2016-5105: scsi: megasas: stack information leakage while reading configuration

```
-     uint8_t data[4096];
+     uint8_t data[4096] = { 0 };
```

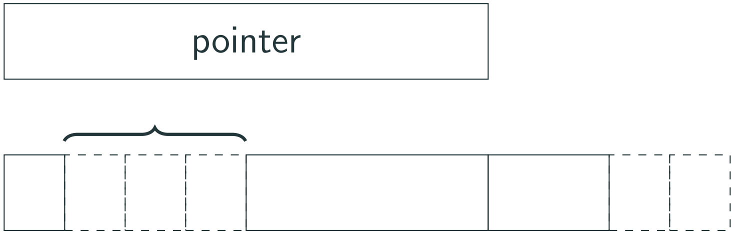
Structure padding



Structure padding



Structure padding



Linux example

Linux example

CVE-2016-4569: infoleak in Linux sound module

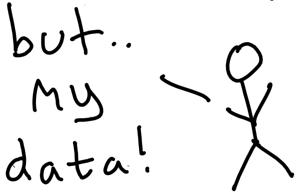
Linux example

CVE-2016-4569: infoleak in Linux sound module

```
struct snd_timer_tread tread;
+ memset(&tread, 0, sizeof(tread));
```

ZERO ALL THE THINGS

MAGIC



safeinit

SafeInit

Goal: ensure initialization of the **heap** and **stack**

SafeInit

Goal: ensure initialization of the **heap** and **stack**

- Custom allocator (heap)
- Compiler pass

SafeInit

Goal: ensure initialization of the **heap** and **stack**

- Custom allocator (heap)
- Compiler pass

```
clang -fsanitize=safeinit
mycode.c
```

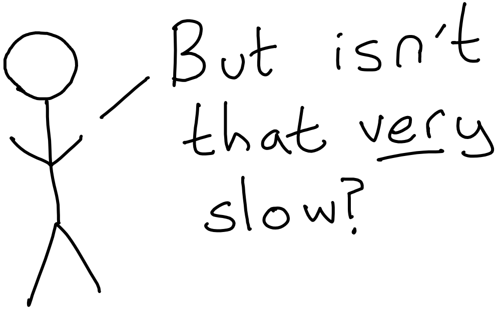
SafeInit

Goal: ensure initialization of the **heap** and **stack**

- Custom allocator (heap)
- Compiler pass

```
clang -fsanitize=safeinit
mycode.c
```

- Using clang/LLVM (May 2016) and `tcmalloc`


But isn't that very slow?

SafeInit stack

```
int a;
```

```
a = 1;
```

SafeInit stack

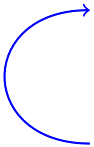
```
int *a = alloca;



*a = 1;
```

SafeInit stack

```
int *a = alloca;

*a = 0;

*a = 1;
```

SafeInit stack

```
int *a = alloca;

*a = 0;

*a = 1;
```



SafeInit stack

`int *a = alloca;`

~~`*a = 0;`~~

`*a = 1;`

```
main() {
  int val; // uninitialized!

  printf("%d", val);
}
```

SafeInit stack: IR-level

baseline

```
define @main() {
  %ptr = alloca i32



  %val = load %ptr
  call printf(..., %val)
}
```

SafeInit stack: IR-level

mem2reg

```
define @main() {
  %ptr = alloca i32



  %val = undef
  call printf(..., %val)
}
```

SafeInit stack: IR-level

mem2reg

```
define @main() {






  call printf(..., undef)
}
```

SafeInit stack: IR-level

baseline

```
define @main() {
  %ptr = alloca i32



  %val = load %ptr
  call printf(..., %val)
}
```

SafeInit stack: IR-level

SafeInit

```
define @main() {
  %ptr = alloca i32
  call llvm.memset(%ptr, 0, 4) // zero it!

  %val = load %ptr
  call printf(..., %val)
}
```

SafeInit stack: IR-level

SafeInit

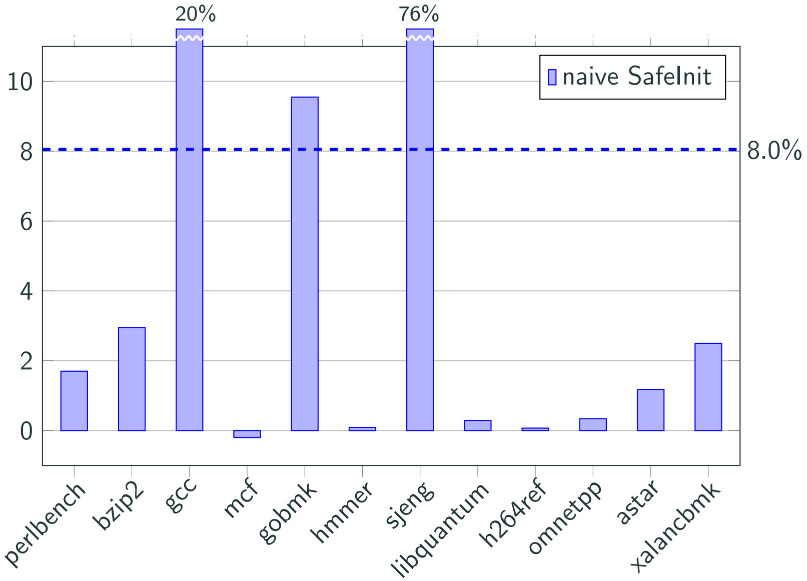
```
define @main() {
  %ptr = alloca i32
  call llvm.memset(%ptr, 0, 4) // zero it!

  %val = 0
  call printf(..., %val)
}
```

SafeInit stack: IR-level

SafeInit

```
define @main() {





  call printf(..., 0)
}
```

SPEC CINT2006: naive SafeInit



optimizations

Sinking stores

```
int a, b, c;

...

if (err) {
  char buf[10000];

  complexPrepare(buf);
  complexError(buf);
}
```

Sinking stores

```
int a, b, c;        ← not here!

...

if (err) {
  char buf[10000];

  complexPrepare(buf);
  complexError(buf);
}
```

Sinking stores

```
int a, b, c;

...

if (err) {
  char buf[10000];  ← ☆here!☆

  complexPrepare(buf);
  complexError(buf);
}
```

More optimizations

- New: Write-only buffer detection

More optimizations

- New: Write-only buffer detection
- Dead Store Elimination patches

More optimizations

- New: Write-only buffer detection
- Dead Store Elimination patches
    - New: Non-constant lengths

More optimizations

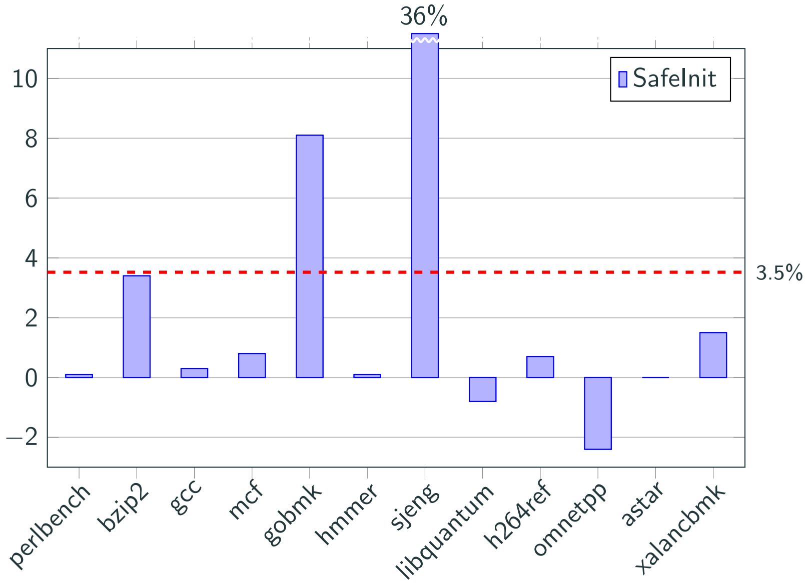
- New: Write-only buffer detection
- Dead Store Elimination patches
    - New: Non-constant lengths
    - New: Non-constant initialization

More optimizations

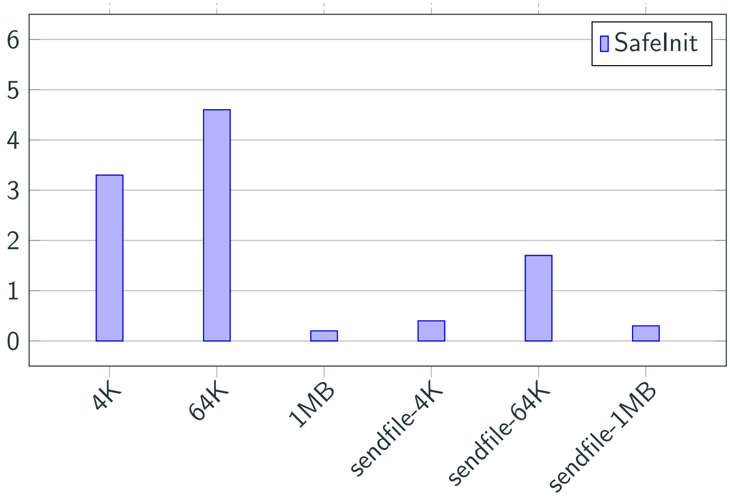
- New: Write-only buffer detection
- Dead Store Elimination patches
    - New: Non-constant lengths
    - New: Non-constant initialization
- More: ‘Safe’ string buffers, non-constant length store shortening, . . .

benchmarks

SPEC CINT2006: SafeInit overhead



nginx: (worst-case) SafeInit overhead



Linux: SafeInit overhead

Worst results from LMbench (microbenchmarking system calls) on LLVMLinux:

Sub-benchmark	SafeInit (stack)
no-op system call	0%
`fstat` system call	4.9%
signal handler (prot fault)	5.9%

Bonus: actually works

- Often just running valgrind is enough to make it obvious we fixed code!
- Some less trivial CVEs: 2016-4243 (PHP), 2016-5337 (qemu), 2016-4486 (Linux)
- Assembly code sometimes changes drastically!

Summary

- **SafeInit**: mitigate this entire class of vulnerabilities by simply guaranteeing initialization on stack and heap

Summary

- **SafeInit**: mitigate this entire class of vulnerabilities by simply guaranteeing initialization on stack and heap
- We obtained acceptable overhead ($< 5\%$)
- Even better as compiler optimizations improve!

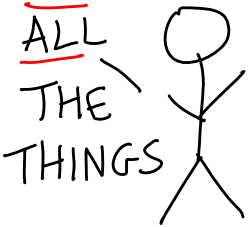
Summary

- **SafeInit**: mitigate this entire class of vulnerabilities by simply guaranteeing initialization on stack and heap
- We obtained acceptable overhead ($< 5\%$)
- Even better as compiler optimizations improve!
- SafeInit prototype: `github.com/vusec/safeinit`

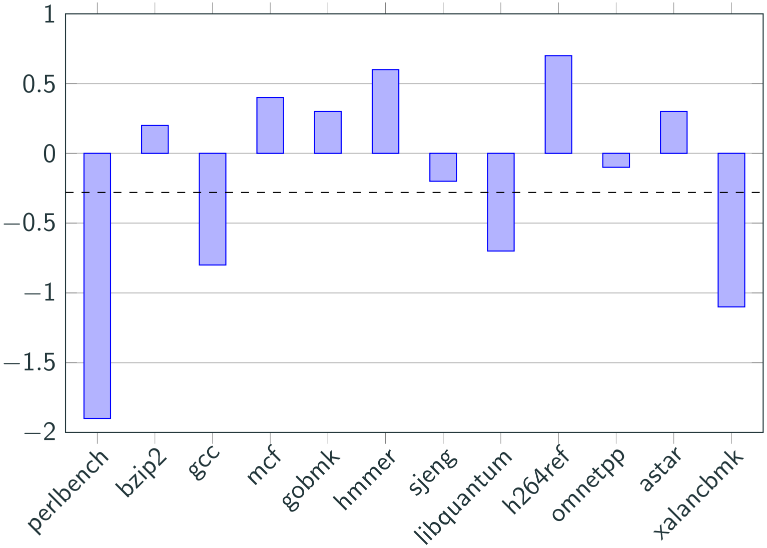
Summary

- **SafeInit**: mitigate this entire class of vulnerabilities by simply guaranteeing initialization on stack and heap
- We obtained acceptable overhead ($< 5\%$)
- Even better as compiler optimizations improve!
- SafeInit prototype: `github.com/vusec/safeinit`
- See the paper for more results and discussion!

Questions?



SPEC CINT2006: Optimizer overhead



Loop initialization

```
int buffer[N];
memset(buffer, 0, sizeof(buffer));

for (int i = 0; i < N; ++i)
  buffer[i] = 1;
```

Loop initialization

```
int buffer[N];
memset(buffer, 0, sizeof(buffer));
initialized(buffer, 0, sizeof(buffer));
for (int i = 0; i < N; ++i)
  buffer[i] = 1;
```

Loop initialization

```
int buffer[N];
~~memset(buffer, 0, sizeof(buffer));~~
initialized(buffer, 0, sizeof(buffer));
for (int i = 0; i < N; ++i)
  buffer[i] = 1;
```

String buffers

```
char buffer[500];
strcpy(buffer, tempString);
strcpy(output, buffer);
```

Undefined behaviour

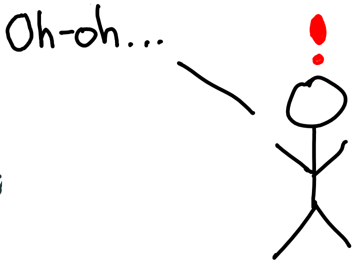
```
int deny_access;

if (deny_access) exit();

print_secret_keys();
```

Undefined behaviour

```
int deny_access = 0;

if (deny_access) exit();

print_secret_keys();
```

Undefined behaviour

```
int deny_access = 0;

~~if (deny_access) exit();~~

print_secret_keys();
```



Annotations

Huge zero initialization getting you down?

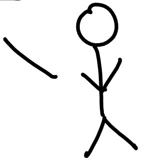
```
__attribute__((no_zeroinit))
```

Annotations

Huge zero initialization getting you down?

```
__attribute__((no_zeroinit))
```

but you said automated!

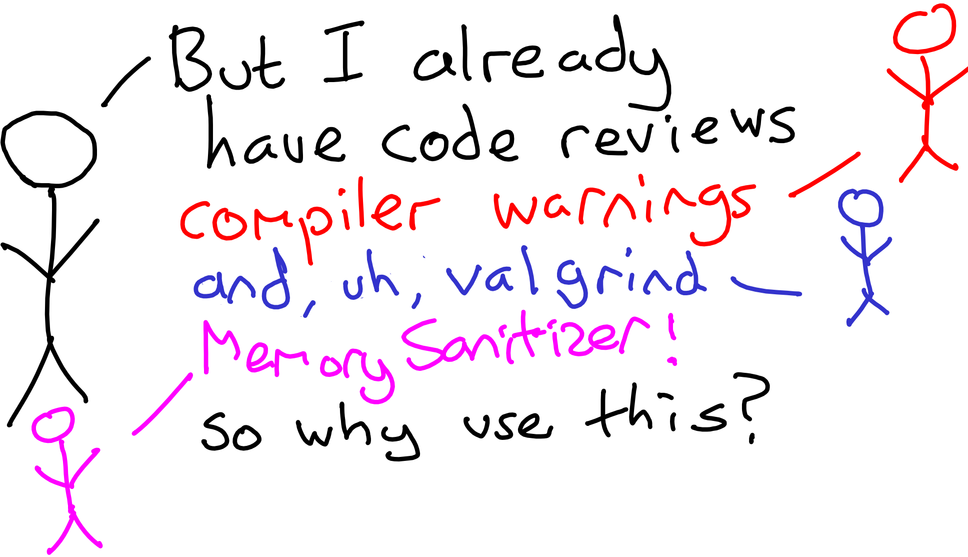


Annotations

Huge zero initialization getting you down?

`__attribute__((no_zeroinit))`

WARNING: Excessive size stack allocation of type `move_s[500]` in `test.c` on line 20


But I already have code reviews
compiler warnings
and, uh, valgrind
MemorySanitizer!
so why use this?

Related Work: mitigation

Heap allocators

Debug allocators, `jemalloc`, ...

Related Work: mitigation

Secure deallocation

“Shredding your garbage”

by Chow et al.

Clear heap memory in `free`!

Related Work: mitigation

Secure deallocation

Frame clearing in epilogue:
"10% – 40%" runtime overhead
(we tried clearing in prologue:
still $> 10\%$)

Related Work: mitigation

PaX gcc plugins

- Stackleak
- Structleak

Related Work: mitigation

UniSan

Lu et al, CCS 2016.

Kernel info exposure:

static analysis + initialization